Comprehensive Path-sensitive Data-flow Analysis

Aditya Thakur

July 5, 2008

Contents

Abstract

Data-flow analysis is an integral part of any aggressive optimizing compiler. We propose a framework for improving the precision of data-flow analysis in the presence of complex control-flow. We initially perform data-flow analysis to determine those control-flow merges which cause the loss in data-flow analysis precision. The control-flow graph of the program is then restructured such that performing data-flow analysis on the resulting restructured graph gives more precise results. The proposed framework is both simple, involving the familiar notion of product automata, and also general, since it is applicable to any forward or backward data-flow analysis. Apart from proving that our restructuring process is correct, we also show that restructuring is effective in that it necessarily leads to more optimization opportunities.

Furthermore, the framework handles the trade-off between the increase in data-flow precision and the code size increase inherent in the restructuring. We show that determining an optimal restructuring is NP-hard, and propose and evaluate a greedy heuristic.

The framework has been implemented in the Scale research compiler, and instantiated for the specific problems of Constant Propagation and Liveness analysis. On the SPECINT 2000 benchmark suite we observe an average speedup of 4% in the running times over Wegman-Zadeck conditional constant propagation algorithm and 2% over a purely path profile guided approach for Constant Propagation. For the problem of Liveness analysis, we see an average speedup of 0.8% in the running times over the baseline implementation.

Acknowledgements

First of all, I would like to thank Prof. Govindarajan for his guidance and infinite patience. I would also like the thank Prof. Uday Khedker for his timely discussions.

I would also like to thank, in order of appearance, Mangesh, Kaushik, Govind, GVSK, Rajini, Mani, Subhajit, Rajesh, Rajan, Kapil, Nandu, Sujit, Ganesh, Girish, Santosh, Mrugesh, Sripathi, Abhishek,. . ..

Finally, I would like to thank my family for their support and encouragement.

Chapter 1

Introduction

Compiler advances double computing power every 18 years.

Proebsting's Law
Todd Proebsting

-"Hey! You! Why do you keep scratching yourself, eh?"
-"Because no one else knows where it itches"

The Benny Hill Show
Benny Hill

It is becoming increasingly difficult to get performance improvement using compiler optimizations, as epitomized by Proebsting's Law [18]. But developers still want that extra 5-15% improvement in the running times of their applications, and compiler optimizations are a safer alternative to manual optimizations carried out by developers which might introduce errors [19].

Data-flow analysis [1] is an integral part of any aggressive optimizing compiler. Information gathered using data-flow analysis is used in code optimizations such as constant propagation, dead-code elimination, common sub-expression elimination, to name a few. Data-flow analysis uses a finite abstraction of the program states and involves computing a fixed-point solution for a set of equations obtained from the control-flow graph of the program. The results of this analysis are used to guide compiler optimizations. Thus, the analysis should be safe in order for the resulting optimizations to be safe.

Imprecision in data-flow analysis leads to a reduction in optimization opportunities. The loss of data-flow precision occurs due to the approximation or merging of differing data-flow facts along incoming edges of a control-flow merge. In the traditional formulation of data-flow analysis, the control-flow graph of the program does not change.

In this thesis, we present a new framework to overcome this imprecision. We initially perform data-flow analysis to determine those control-flow merges that cause the loss in data-flow analysis precision, which we call *Destructive Merges*. The control-flow graph (CFG) of the program is then restructured to remove the effects of such destructive merges. Performing data-flow analysis on the resulting restructured graph gives more precise results, and effectively eliminates the destructive merges. This leads to more optimization opportunities in the resulting CFG. Thus, we see that we use the results of an initial data-flow analysis to restructure the CFG and improve the precision of the data-flow analysis on the resulting CFG [1].

The framework presented in this thesis is simple and clean, and uses the familiar notion of product automata [13] to carry out the restructuring transformation. Further, the framework is general since it can be instantiated to any forward or backward data-flow analysis. This clean formulation allows us to show that the restructuring is correct, in the sense that the original and restructured CFGs are equivalent. Also, we prove that the restructuring is profitable in that it necessarily leads to more optimization opportunities in the restructured CFG.

The framework we develop is not only applicable to all forward data-flow analysis such as constant propagation, points-to analysis but also to backward data-flow analysis such as liveness and anticipatability. We believe that a compiler writer would benefit from an approach based on sound theorotical guarantees which is clean and simple to implement.

The restructuring inherently entails an increase in code size due to code duplication i.e., the increase in precision comes at the cost of an increase in code size. Furthermore, we show that determining an optimal restructuring is NP-Hard and, hence, propose a greedy heuristic for restructuring. Our framework explicitly handles the trade-off between precision and code size increase and also makes use of low-cost basic-block profile information.

We have implemented the proposed framework in the Scale research compiler [21] and instantiate it with the specific problem of Constant Propagation and Liveness analysis [1]. We compare our technique with the Wegman-Zadeck conditional constant propagation algorithm [25](*Base*) and with a purely path profile-guided restructuring technique [2](*HPG*). On the SPECINT 2000 benchmark suite [22], our technique exposes, on an average, *3.5 times more dynamic constants* as compared to the HPG technique. Further, we observe an average speedup of *4%* in the running times over Base and *2%* over the HPG technique. On the other hand, for Liveness analysis

[1]Hence, the Benny Hill quote!

we find that our approach gives an average speedup of 0.8% with an average code size increase of 15%.

Contributions

These are the main contributions of this thesis.

- The framework can be instantiated to any forward or backward dataflow analysis.
- The restructuring algorithm seamlessly handles complex control flows in a program like nested loops, switch cases and so on.
- We provide provable guarantees about correctness of the algorithm. Furthermore, we guarantee an increase in optimization opportunities in the restructured program.
- We prove that obtaining the optimal restructured program is NP-Hard and propose heuristics to solve the problem efficiently.
- The framework has been implemented in the *Scale* compiler and is found to be better than a purely profile driven approach.

Thesis Outline

The rest of the thesis is organised as follows:

Chapter 1 *Preliminaries* In this chapter we introduce the notations used in the rest of thesis. Specifically, we define out notion of destructive merges. We also provide an algorithm which provides an overview of our approach.

Chapter 2 *Transformation* In this chapter, we describe our restructuring transformation for both forward and backward analysis.

Chapter 3 *Trade-off* Since our approach inherently entails an increase in code size, we have to deal with the trade-off between the increase in precision and increase in code size. In this chapter, we prove that getting the optimal restructured graph is NP-Hard and explain a greedy algorithm to tackle this problem.

Chapter 4 *Related Work* This chapter provides an overview of the related literature pertaining to improving data-flow analysis precision. Specifically, it discussses the Hot Path Graph approach

Chapter 5 *Experimental Results* Having explained the theoretical framework, we show our empirical results for the specific problems of Constant Propagation and Liveness analysis.

Chapter 6 *Conclusions and Future Directions* We suggest further directions in which this approach can be extended and conclude the thesis.

Chapter 2

Preliminaries

I can go with the flow,
But don't say it doesn't matter, matter
anymore.
I can go with the flow,
Do you believe it in your head?

Go With The Flow
QUEENS OF THE STONE AGE

In this chapter, we introduce the notations used in the rest of the thesis. We start with basic definitions of control flow graphs followed by definitions of both backward and forward data-flow analysis. Using these notions we define *destructive merges*, a concept which plays a central part in our approach. Finally, we discuss concepts from automata theory which are used to carry out the restructuring.

2.1 Control-flow graphs

Definition 1. (SIMPLE GRAPH) *A* simple graph *is a graph that does not have more than one edge between any two vertices and no edge starts and ends at the same vertex.*

Definition 2. (CONTROL FLOW GRAPH) *A* control flow graph (CFG) $G = (N, E, \text{start}, \text{end})$ *is a simple directed graph* (N, E) *where*

- *the nodes N represent program statements of a function,*
- *the edges E represent possible control flow between program statements,*
- *the* start *node has no incoming edges and all nodes are reachable from the* start *node*

- end *node has no outgoing edges and the* end *node is reachable from all other nodes.*

For simplicity of exposition we consider the nodes to be individual program statements, but they could also be basic blocks.

Let $\texttt{pred}(n)$ denote the set of control-flow predecessors of a node n in the CFG, and $\texttt{succ}(n)$ denote the set of control-flow successors of a node n in the CFG. By definition, $\texttt{pred}(start) = \emptyset$, and $\texttt{succ}(end) = \emptyset$.

Definition 3. (Control Flow Paths) *A* path *in a control-flow graph is a sequence of nodes or edges. As is usually assumed in data-flow analysis, all possible paths in the CFG are considered to be feasible.*

Definition 4. (Reachable Nodes) *Given a node n* reachable nodes *i.e.* $\texttt{reachable\_nodes}(n)$ *is the set of nodes reachable from node n along some path in the CFG.*

Definition 5. (Generalized Post-dominator Set) *A set of edges W* post-dominate *vertex v, i.e.* $\texttt{postdom}(v) = W$ *iff the following two conditions are met:*

1. *all paths from the vertex v to the* end *node contain some edge $w \in W$; and*
2. *for each edge $w \in W$, there is at least one path from vertex v to the* end *node which contains w and does not contain any other edge in W.*

Example. Figure 2.1 shows the control-flow graph G.

$$\texttt{pred}(start) = \emptyset,\ \texttt{succ}(end) = \emptyset,$$

$$\texttt{pred}(D) = \{B, C\},\ \texttt{succ}(D) = \{E, F\},$$

$$\texttt{postdom}(A) = \{(A, B), (A, C)\}, \texttt{postdom}(D) = \{(D, E), (D, F)\}.$$

□

Definition 6. (Reverse Graph) *Given a simple directed graph G , the* Reverse Graph *is a graph in which the direction of all edges have been reversed.*

Definition 7. (Backward Reachable Nodes) *Given a node n in CFG G,* backward reachable nodes *i.e.* $\texttt{backward\_reachable\_nodes}(n)$ *is the set of nodes reachable from node n along some path in the* reverse graph *of CFG G.*

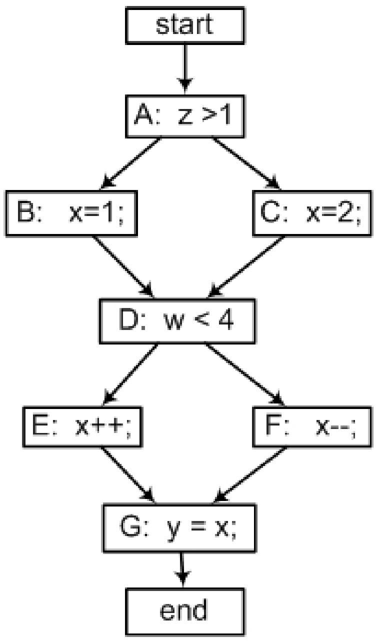


Figure 2.1: Control Flow Graph G.

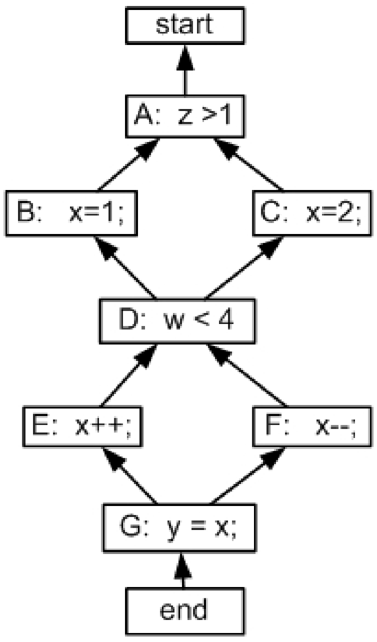


Figure 2.2: The Reverse Graph R corresponding to the CFG G in Figure 2.1

Example. Figure 2.2 shows the reverse graph R of the CFG in Figure 2.1.
In graph R, `reachable_nodes`$(D) = \{start, A, B, C, D\}$.
Hence, in graph G, `backward_reachable_nodes`$(D) = \{start, A, B, C, D\}$.

$\square$

The program statements in a CFG usually consist of conditional statements such as *if* statements, and assignment statements. We will also make use of *assume* statements in our control-flow graphs. An *assume* statement takes a predicate, say p, as an argument. An assume statement adds an assumption about the predicate expression. For example, $assume(y < 3)$ would restrict y to the values less than 3 in all subsequent computation. Control can never go past the *assume* statement if the predicates evaluates to false (F). If the predicate p evaluates to true (T), then the program continues. But in case the predicate cannot be evaluated then we effectively assume that the predicate is true and continue. This might happen if the variables involved in the *assume* have not been assigned to. Thus,*assume* statements control the flow of the program but do not modify the program state. These *assume* statements are commonly seen in work related to verification techniques, but we will find them useful when discussing backward analysis.

Such *assume* statements can replace *if* statements, viz.

$$if(e)\ then\ A;\ \ else\ B;$$

can be written as

$$if(*)\ then\ assume(e);\ \ A;\ \ else\ assume(\neg e);\ \ B;.$$

Though at first this might seem nondeterministic, it is easy to still maintain deterministic execution since the *assume* statements here are along the immediate successors of the *if* statement.

The predicate of the *assume* statement seen along an edge is called the *edge predicate* for the edge.

Example. Figure 2.3 shows the CFG G of Figure 2.1 with the *if* statement $if(z > 1)$ at node A are replaced by corresponding *assume* statements $assume(z > 1 = T)$ and $assume(z > 1 = F)$ along outgoing edges (A, B) and (A, C) respectively.
`edge_predicate`$((A, B)) = (z > 1 = T)$, (`edge_predicate`$((A, C)) = (z > 1 = F)$

$\square$

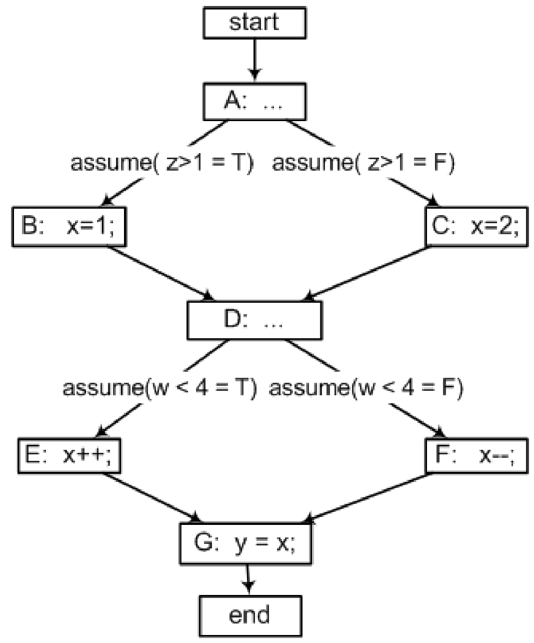


Figure 2.3: The CFG G of Figure 2.1 in which the *if* statements at nodes A and D are replaced by corresponding *assume* statements along their outgoing edges.

2.2 Data-flow Analysis

Data-flow analysis [1] is a static analysis technique used to glean information about the program as a whole and it's possible behaviors. It forms the core of any aggressive compiler optimization framework. It is also used extensively in program understanding and verification tools.

Any data-flow anlysis should be *safe* i.e. it should be conservative and over-approximate the runtime behaviour of the program. This ensures that the code transformations and optimization carried out using the analysis results do not change the semantic meaning of the program. A typical over-approximation which is carried out is that all control-flow paths in the CFG are assumed to be possible, even if some are infeasible. An analysis which does not consider a control-flow path which could be taken at runtime is deemed unsafe.

The Control Flow Graph (CFG) is the principle abstraction of the program control-flow over which data-flow analysis operates. In terms of the actual information, data-flow analysis operates over a finite abstraction of the possible program states, which is usually represented as a complete semi-lattice. We assume that the data-flow information is associated with the *in* and *out* of a node, where for a given node n, *in* is the program point immediately preceding the node n, and *out* is the program point immediately following the node n. Data-flow information is collected by setting up a system of equations which relate information at various points in a program and finding the fixed point of these equations. For example, the data-flow information at *in* of a node is related to that at the *out* of it's predecessor node.

We begin with some common concepts of data-flow analysis adopted from [1, 2].

Definition 8. (Data-flow Analysis Framework) *A Data-flow analysis framework $\mathcal{F}$ is a tuple $(L, \wedge, F, \top, \mathcal{D})$ where:*

- *L is a complete semilattice with the meet operation $\wedge$.*
- *F is a set of monotonic transfer functions from L to L.*
- *$\top$ is the top element in L, satisfying the law $\top \wedge \mu = \mu$ for all μ in L,*
- *$\mathcal{D}$ is the direction of analysis, either* forward *or* backward*.*

Definition 9. (Data-flow Problem) *A data-flow problem $\mathcal{P}$ is a tuple $(G, \mathcal{F}, l_r, M)$ where:*

- *$G = (N, E, \text{start}, \text{end})$ is a control-flow graph (CFG),*
- *$\mathcal{F}$ is a data-flow analysis framework,*
- *$l_s \in L$ is the data-flow fact associated with* start*,*
- *$M : N \to F$ maps the nodes of G to the functions in F of $\mathcal{F}$.*

M can be extended to paths by composition i.e.

$$M([n_1, n_2, \ldots, n_k]) \overset{\text{def}}{=} M(n_k) \circ M(n_{k-1}) \circ \cdots \circ M(n_2) \circ M(n_1)$$

The following definitions assume that the direction of the analysis is *forward*, and can be naturally extended to *backward* analysis.

Definition 10. (Data-flow Solution) *A solution $\mathcal{S}$ of $\mathcal{P}$ is a map $\mathcal{S} : N \to L$ such that, for any path p from* start *to node u, $\mathcal{S}(u) \preceq (M(p))(l_s)$.*

The solution is associates data-flow facts with the *in* of the node n, which we represent as $\texttt{in\_dff}(n)$. The corresponding data-flow facts at the *out* of a node n, i.e. $\texttt{out\_dff}(n)$, can be determined by applying the node's transfer function to $\texttt{in\_dff}(n)$. Thus,

$$\texttt{out\_dff}(n) \overset{\text{def}}{=} M(n)(\texttt{in\_dff}(n)).$$

Definition 11. (Fixed Point Solution) *A* fixed point $\widehat{\mathcal{S}}$ *of* $\mathcal{P}$ *is a map* $\widehat{\mathcal{S}} : N \rightarrow L$ *such that* $\widehat{\mathcal{S}}(\text{start}) \preceq l_s)$ *and, if* e *is an edge from node* u *to node* v*,* $\widehat{\mathcal{S}}(v) \preceq (M(u))(\widehat{\mathcal{S}}(u))$*.*

Definition 12. (Maximum Fixed Point Solution) *A* Maximum Fixed Point (MFP) Solution $\mathcal{S}$ *of* $\mathcal{P}$ *is a solution of* $\mathcal{P}$ *such that, for any fixed point* $\widehat{\mathcal{S}}$*,* $\widehat{\mathcal{S}}(u) \preceq \mathcal{S}(u)$ *for all* $u \in V$*.*

The MFP solution can be computed by general iterative algorithms [1, 15].

2.2.1 Constant Propagation

In the Constant Propagation problem we are interested in knowing whether a variable x always has a constant value at a program point. The *use* of the variable at that point can then be replaced by that constant value. Dead-code elimination, common sub-expression elimination, redundancy elimination are some of the other analysis which benefit from precise constant propagation analysis.

Specifically, in the Constant Propagation data-flow analysis framework $\mathcal{C}$, a variable x can have any one of the three data-flow facts associated with it: (1) $\top$, which means that nothing has been asserted about x, (2) c_i, which means that x has a constant value, (3) $\perp$, which means that x has been determined to not have a constant value.

L can be thought of as a vector of data-flow values, one component for each variable in the program. Given $\ell \in L$ and a variable x, we will use the notation $\ell[x]$ to denote the data-flow value associated with x in ℓ. For example, $\mathcal{S}(u)[x]$ represents the data-flow value of variable x at node $u \in V$ given a solution $\mathcal{S}$ to the Constant Propagation problem $\mathcal{C}$.

Example. Consider Figure 2.4. We would like to know whether the *use* of the variable x at node G can be replaced by a contant value. Constant Propagation analysis would ascertain that this cannot be done.

□

2.2.2 Liveness Analysis

A variable is said to be *live* at a particular point in a program if there is a path to the exit along which its value may be used before it is redefined. It is *Not live* if no such path exists. The values live (L) and not live (N) form a lattice where $\top = N$ and $\perp = L$. Liveness analysis is a backward analysis.

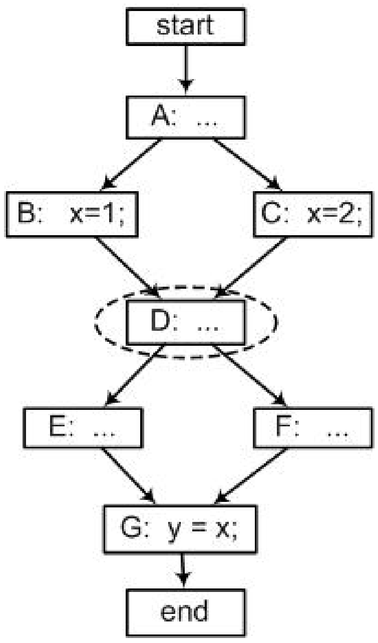


Figure 2.4: A simple acyclic program. Node D is a destructive merge. The *use* of variable x at node G cannot be replaced by a constant.

2.2.3 Anticipatability Analysis

An expression's value is said to be *anticipatable* on entry to block i if every path from that point includes a computation of the expression and if placing that computation at any point on these paths produces the same value.

2.3 Destructive Merge

2.3.1 Forward Analysis

Central to our approach, is the notion of a destructive merge. Intuitively, a destructive merge is a control-flow merge where data-flow analysis loses precision. The notion of precision is governed by the partial-order of the lattice of the data-flow problem.

Definition 13. (Destructive Merge) *For a given forward data-flow problem and its corresponding solution, a control-flow merge m is said to be a* Destructive Merge *if* $\exists p \in \texttt{pred}(m)$ *s.t.* $\texttt{in\_dff}(m) \prec \texttt{out\_dff}(p)$.

Definition 14. (Destroyed Data-flow Facts) *Given a destructive merge m, a data-flow fact d is said to be a* Destroyed Data-flow Fact , *i.e.*

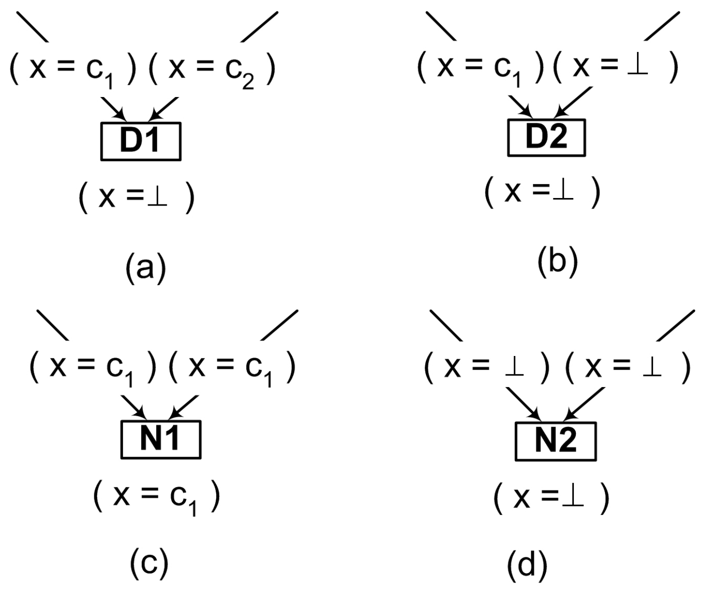


Figure 2.5: For the problem of Constant Propagation, (a)-(b) show two destructive merges and (c)-(d) illustrate two control-flow merges which are not destructive. c_1 and c_2 are two differing constants, while $\perp$ represents not-constant.

$d \in \texttt{destroyed\_dff}(m)$, iff $d \in \texttt{out\_dff}(p)$, where node $p \in \texttt{pred}(m)$, and $\texttt{in\_dff}(m) \prec d$.

Example: For the problem of Constant Propagation, Figure 2.5 illustrates the different scenarios possible at a control-flow merge. Nodes $D1$ and $D2$ are destructive, while nodes $N1$ and $N2$ are not.

More specifically, in Figure 2.4 node D is a destructive merge since $\texttt{in\_dff}(D) = \{\perp\}$, while $\texttt{out\_dff}(B) = \{\texttt{x} = \texttt{1}\}$. Further,

$$\texttt{destroyed\_dff}(m) = \{\texttt{x} = \texttt{1}, \texttt{x} = \texttt{2}\}.$$

□

2.3.2 Backward Analysis

The above definitions can be easily extended to Backward data-flow analysis. Instead of control-flow merges, destructive merges for backward analysis defined over control-flow forks.

Definition 15. (Destructive Merge) *For a given backward data-flow problem and its corresponding solution, a control-flow fork m is said to be a* Destructive Merge *if $\exists p \in \texttt{succ}(m)$ s.t. $\texttt{out\_dff}(m) \prec \texttt{in\_dff}(p)$.*

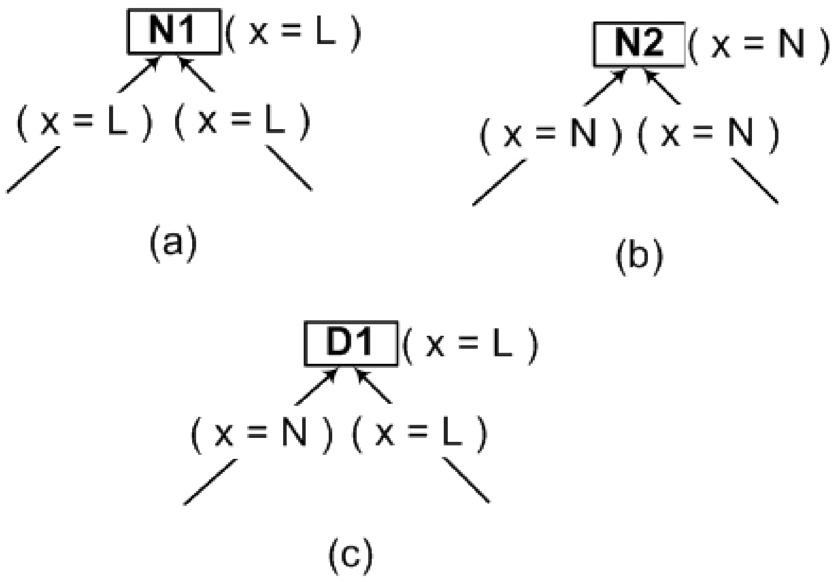


Figure 2.6: For the problem of Liveness Analysis, (a) shows a destructive merge and (b)-(c) illustrate two control-flow merges which are not destructive.

Definition 16. (DESTROYED DATA-FLOW FACTS) *Given a destructive merge m, a data-flow fact d is said to be a* Destroyed Data-flow Fact *, i.e. $d \in \texttt{destroyed\_dff}(m)$, iff $d \in \texttt{in\_dff}(p)$, where node $p \in \texttt{succ}(m)$, and $\texttt{out\_dff}(m) \prec\ d$.*

Example. For the problem of Liveness Analysis, Figure 2.6 illustrates the different scenarios possible at a control-flow fork. Node $D1$ is destructive, while nodes $N1$ and $N2$ are not.

More specifically, in Figure 2.4 node D is a destructive merge since $\texttt{out\_dff}(D) = \{\texttt{x} = \texttt{L}\}$, while $\texttt{in\_dff}(B) = \{\texttt{x} = \texttt{N}\}$. Further,

$$\texttt{destroyed\_dff}(m) = \{\texttt{x} = \texttt{L}\}.$$

□

2.4 Automata Theory

As we will see our restructuring algorithm uses many concepts from automata theory, which we introduce next.

Definition 17. (DETERMINISTIC FINITE AUTOMATON) *A* deterministic finite automaton *is defined by the tuple* $(Q, \Sigma, \delta, s, F)$ *where*

- Q *is a finite of set, the elements of the set are called* states*,*
- Σ *is a finite set, the* input alphabet*,*
- $s \in Q$ *is a unique* start state*,*
- $F \subseteq Q$*, the elements of* F *are called the* final *or* accept states*,*
- $\delta : Q \times \Sigma \to Q$ *is the* transition function*. Intuitively,* δ *is a function that tells which state to move to in response to an input.*

Unless mentioned otherwise, all finite automata discussed in this thesis are *deterministic.*

Definition 18. (MULTISTEP TRANSITION FUNCTION) *Given a finite automaton* $M(Q, \Sigma, \delta, s, F)$ *we defined the* multistep transition function $\widehat{\delta} : Q \times \Sigma^* \to Q$ *inductively on the length of* x

$$\widehat{\delta}(q, \epsilon) \overset{\texttt{def}}{=} q,$$

$$\widehat{\delta}(q, xa) \overset{\texttt{def}}{=} \delta(\widehat{\delta}(q, x), a).$$

The function $\widehat{\delta}$ maps a state q and a string x to a new state $\widehat{\delta}(q, x)$. Using $\widehat{\delta}$ we can now define the *language* of a finite automaton.

Definition 19. (LANGUAGE OF FINITE AUTOMATON) *The set or* language *accepted by a finite automaton* M *is the set of strings accepted by* M *and is denoted* $L(M)$:

$$L(M) \overset{\texttt{def}}{=} \{x \in \Sigma^* \mid \widehat{\delta}(s, x) \in F\}.$$

There are many operations which can be defined over finite automata. The one which we are most interested in is the *product operator* $\times$.

Definition 20. (PRODUCT AUTOMATON) *Given two finite state automata* $M_1(Q_1, \Sigma, \delta_1, s_1, F_1)$ *and* $M_2(Q_2, \Sigma, \delta_2, s_2, F_2)$*, the* product *of* M_1 *and* M_2 *denoted by* $M_1 \times M_2$ *is defined as the finite state automaton* $M_3(Q_3, \Sigma, \delta_3, s_3, F_3)$ *where*

- $Q_3 = Q_1 \times Q_2 = \{(p, q) \mid p \in Q_1 \text{and } q \in Q_2\}$,
- $F_3 = F_1 \times F_2 = \{(p, q) \mid p \in F_1 \text{and } q \in F_2\}$,
- $s_3 = (s_1, s_2)$,
- $\delta_3 : Q_3 \times \Sigma \to Q_3$*, the transition function is defined as*

$$\delta_3((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$$

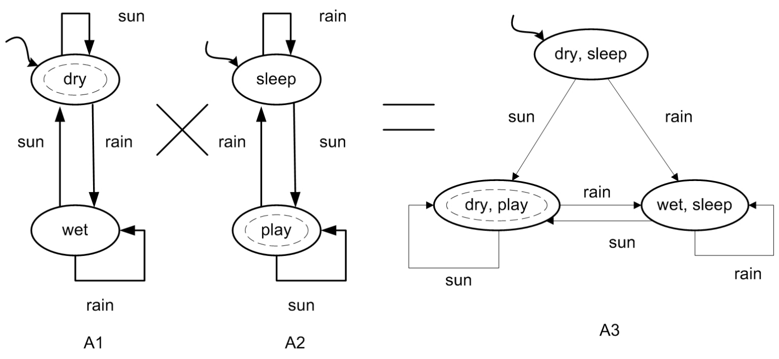


Figure 2.7: Figure illustrating the product operation for finite automata.

Example. Figure 2.7 shows two automata A_1 and A_2 whose alphabet is $\Sigma = \{\text{rain}, \text{sun}\}$. The start and accept state for A_1 is state *dry*, while the start state for A_2 is *sleep* and accept state is *play*. The transition function is shown graphically in the figure.

The product automaton $A_3 \overset{\texttt{def}}{=} A_1 \times A_2$ is also shown.

$\square$

Further, since the product operation is both associative and commutative, we can define the product operator for k in a natural fashion as follows:

Definition 21. (Generalized Product Automaton) *The* product *of* k *finite state automata* $M_1(Q_1, \Sigma, \delta_1, s_1, F_1)$, $M_2(Q_2, \Sigma, \delta_2, s_2, F_2)$, $\ldots$,$M_k(Q_k, \Sigma, \delta_k, s_k, F_k)$ *denoted by* $M_1 \times M_2 \times \ldots \times M_k$ *is defined as the finite state automaton* $M_p(Q_p, \Sigma, \delta_p, s_p, F_p)$ *where*

- $Q_p = Q_1 \times Q_2 \times \ldots \times Q_k = \{(q_1, q_2, \ldots, q_k) \mid q_i \in Q_i, 1 \le i \le k\}$,
- $F_p = F_1 \times F_2 \times \ldots \times F_k = \{(q_1, q_2, \ldots, q_k) \mid q_i \in F_i, 1 \le i \le k\}$,
- $s_p = (s_1, s_2, \ldots, s_k)$,
- $\delta_p : Q_p \times \Sigma \to Q_p$, *the transition function is defined as*

$$\delta_p((q_1, q_2, \ldots, q_k), a) = (\delta_1(q_1, a),\ \delta_2(q_2, a),\ \ldots,\ \delta_k(q_k, a))$$

Though simple to understand and construct, the product operator is extremely useful and enjoys the following property:

Lemma 1. (Intersection Lemma) *Consider two automata* A_1 *and* A_2. *If* $P = A_1 \times A_2$, *then*

$$L(P) = L(A_1) \cap L(A_2).$$

```
PRODUCT( $M_1(Q_1, \Sigma, \delta_1, s_1, F_1),\ M_2(Q_2, \Sigma, \delta_2, s_2, F_2),\ \ldots\ M_k(Q_k, \Sigma, \delta_k, s_k, F_k)$ )
      // The start node of product automaton composed of start nodes of input automata.
 1    $s_P \leftarrow (s_1, s_2, \ldots, s_k)$
      // The states of the product automaton; initially only contains the start state.
 2    $Q_P \leftarrow s_P$
      // The final nodes of the product automaton.
 3    $F_P \leftarrow F_1 \times F_2 \times \ldots \times F_k$
      // Compute the transition function for the product automaton.
      // The transition function of the product automaton; initially empty.
 4    $\delta_P \leftarrow \emptyset$
      // Worklist of states; initially contains the start state.
 5    WorkList $W \leftarrow s_P$
      // Do while the worklist is not empty.
 6    While $W \neq$ EMPTY
 7        do
 8            $state \leftarrow$ get($W$)  // Pop the next state from the worklist.
 9            $(q_1, q_2, \ldots, l_k) \leftarrow state$
              // For each letter in the input alphabet, compute the next state.
10              Foreach $a \in \Sigma$
11                do
                      // Get the next state.
12                    $next\_state \leftarrow (\delta_1(q_1, a),\ \delta_2(q_2, a),\ \ldots,\ \delta_k(q_k, a))$
                      // Check if this is a new state.
13                    If($next\_state \notin Q_P$)
14                      then  // Add the new state to the set of states.
15                            $Q_P \leftarrow Q_P \cup \{\ next_state\ \}$
                              // Push the new state onto the worklist.
16                            put($W, next\_state$)
                      // Add the new transition to the transition function.
17                    $\delta_P \leftarrow \delta_P \cup \{\ (state, next\_state)\ \}$
18    return $M_P(Q_P, \Sigma, \delta_P, s_P, F_P)$
```

Figure 2.8: Algorithm to compute the product automaton

Proof. We refer the reader to [13] for the proof. $\square$

Definition 22. (Control Flow Automaton) *The* Control Flow Automaton $\widehat{G}$ *corresponding to a CFG* $G(N, E, \text{start}, \text{end})$ *is finite automaton* $(Q, \Sigma, \delta, s, F)$ *where*

- *the set of states* $Q \stackrel{\text{def}}{=} N$*,*
- *the input alphabet* $\Sigma = E$*,*
- *the start state* $s \stackrel{\text{def}}{=} \text{start}$*,*
- *the final states* $F = \{\text{end}\}$*,*
- *the transition function* $\delta : Q \times \Sigma \to Q$ *is defined as*

$$\delta(p, a) \stackrel{\text{def}}{=} q \textit{ iff node } p \textit{ is connected to } q \textit{ by the edge } a \textit{ in } G.$$

This natural correspondence between control-flow graphs and finite automaton allows us to use the concepts in automata theory described in Section 2.4. When the context is clear, we will use G to represent both the CFG G and the control flow automaton $\widehat{G}$.

Definition 23. (CFG Equivalence) *We say that the two CFGs* G_1 *and* G_2 *are* equivalent*, i.e.* $G_1 \equiv G_2$*, if the languages accepted by* $\widehat{G_1}$ *is equal to language accepted* $\widehat{G_2}$*, i.e.,* $L(\widehat{G_1}) = L(\widehat{G_2})$*.*

Such a notion of equivalence suffices since the restructuring only duplicates nodes and does not change the semantics of the statements corresponding to a node.

2.5 Overall Algorithm

Figure 2.5 presents an overview of our approach. Central to our approach is the notion of a destructive merge. Having performed data-flow analysis, we determine which data-flow merges are destructive. This was explained in Section 2.3. Since our approach implicitly entails an increase in code size, we need to determine the best set of destructive merges to eliminate. We tackle this issue in Chapter 4. Having determined which destructive merges have to eliminated, the Split Automaton is constructed. Definition 33 tells us how to do this for Forward analysis, while Definition 44 applies for Backward analysis.

```
SPLIT( $G, k$)
1   *Perform Data-flow analysis*
    *//set of destructive merges*
2   $DM \leftarrow \emptyset$
3   **Foreach** merge $m$
4         **do**
5               **If** $m$ is destructive
6                  **then**
7                         $DM \leftarrow DM \cup \{m\}$
8   **Foreach** merge $m \in DM$
9   Calculate *fitness* (m)
10  **Foreach** $k$ fittest non-zero merge $m_i$
11        **do**
12  $A_i = split\_automaton(m_i)$
13  *// Let* $A_1, A_2, ..., A_k$ *be the split automata*
14  $S \leftarrow$ *split_graph* $(G, A_1, A_2, ..., A_k)$
15  Perform Data-flow analysis on $S$
16  Optimize $S$
```

Figure 2.9: The Split Algorithm

The actual restructuring transformation is carried out using these Split Automaton. Definition 35 defines this for Forward analysis, while Definition 46 is to be used for Backward analysis. We show that the restructured graph is equivalent to the original.

Finally, data-flow analysis is carried out on this restructured graph and the final program is optimized.

Chapter 3

The Transformation

Paradoxical as it seemed, the Master always insisted that the true reformer was one who was able to see that everything is perfect as it is - and able to leave it alone.

One Minute Wisdom
ANTHONY DE MELLO

In this chapter we describe the transformation applied to a CFG to eliminate a destructive merge. We will first consider forward data-flow analysis, and then extend the approach to handle backward data-flow analysis. Further, in order to explain the concepts clearly, specific data-flow analysis will be used, before stating the general framework.

We start with a program P and a data-flow analysis D. We have already seen that destructive merges lead to loss of data-flow precision and, hence, result in fewer optimization opportunities. We wish to transform the program P by altering the control-flow to get an equivalent program P' in order to eliminate the effects of such destructive merges. Performing data-flow analysis on the program P' gives more precise results. The transformation described is *clean* and *general* enough to be applied to any data-flow analysis.

This chapter does not explicitly deal with the trade-off between data-flow precision and code size. As we will see in Chapter 4, the increase in code size is determined by the choice of destructive merges which are to be eliminated. For the time being, we assume that the set of destructive merges to be eliminated are given and defer the description of the algorithm to determine this set to Chapter 4. We first describe the solution for forward data-flow analysis, and in particular, we use the specific problem of Constant Propagation (Section 2.2.1) for explanatory purposes before generalising. In Section 3.1.1, we describe an obvious *Naïve* solution to the problem. As we

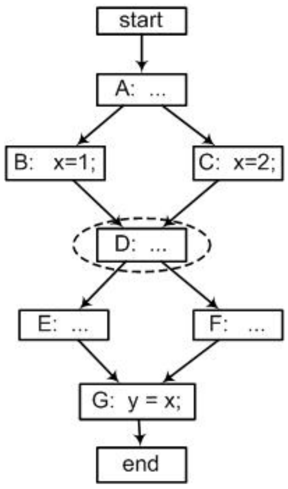


Figure 3.1: Simple acyclic program $P1$. Node D is a destructive merge. *Use* of variable x at node G cannot be replaced by a constant.

will see, this solution is far from satisfactory in terms of its efficiency and increase in code size. We then present our approach and in Section 3.1.2 answer the question of which nodes are duplicated in our approach. Nodes which are not required to be split in order to improve precision are not selected. In Section 3.1.3, we see how these nodes are duplicated and how the CFG is split in order to ensure that the resulting CFG is equivalent and that the data-flow precision improves.

3.1 For Forward Analysis

3.1.1 The Naïve Solution

We illustrate the *Naïve* approach using a specific example and compare it to our approach. Consider the program $P1$ shown in Figure 3.1. Node D is a destructive merge since the data-flow facts $\{\texttt{x} = \texttt{1}\}$ and $\{\texttt{x} = \texttt{2}\}$ are merged to get the data-flow fact $\{\texttt{x} = \texttt{nc}\}$. Thus, the *use* of variable x at node G cannot be replaced by a constant. We wish to improve the data-flow precision by restructuring the control-flow graph. The first approach is to duplicate the destructive merge. Thus, consider the following *Naïve* solution: as long as there exists a destructive merge in the CFG, duplicate the destructive merge. In the transformed graph, there is one copy of the destructive merge for each incoming edge, while the outgoing edges of the copies remain the same as the original. According to this approach, node D is duplicated, and we now get two nodes $D1$ and $D2$, one for each of the incoming edges from

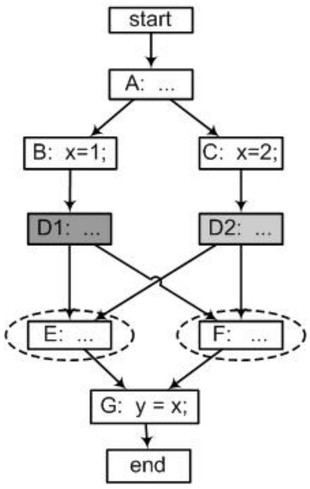


Figure 3.2: Simple acyclic program of Figure 3.1 with node D duplicated. Nodes E and F are destructive merges. *Use* of variable x cannot be replaced by a constant.

nodes B and C respectively. The outgoing edges of nodes $D1$ and $D2$ are the same as that of node D. The resulting program is shown in Figure 3.2.

In this figure, the destructive merge at node D is removed. Yet, we still cannot replace the *use* of variable x at node G. The reason for this is easy to understand. We see that in Figure 3.2, nodes E and F have become destructive merges. The destructive merge at node D in Program $P1$ (Figure 3.1) has simply shifted to nodes E and F in Figure 3.2 as a result of duplicating node D.

If we continue duplicating destructive merges as per the *Naïve* approach, then we obtain the program in Figure 3.4. In this figure, the *use* of variable x at nodes $G1$, $G2$, $G3$ and $G4$ can be replaced by constants 1, 1, 2 and 2 respectively.

Compared to this, our approach directly transforms the CFG of Figure 3.1 to that of Figure 3.5 in a single step. There are two reasons why our approach is better than this *Naïve* approach.

- To identify destructive merges data-flow analysis needs to be performed. Thus, in the *Naïve* approach, each time the CFG is restructured data-flow analysis needs to be performed. In comparison, as we see, our approach performs data-flow analysis only once to identify destructive merges. Then in a single step, the CFG is transformed in such a way that the data-flow precision is guaranteed to improve.

- The code size increase due to this *Naïve* approach might be unneces-

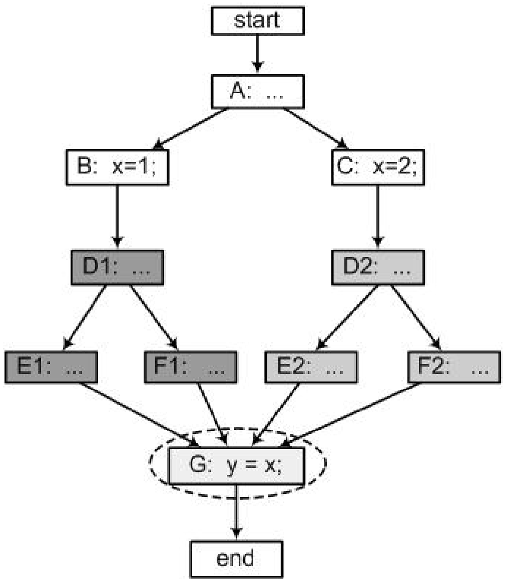


Figure 3.3: Naïve restructuring of the acyclic program of Figure 3.2. in which nodes E and F are duplicated. Node G is a destructive merge and the *use* of variable x cannot be replaced by a constant.

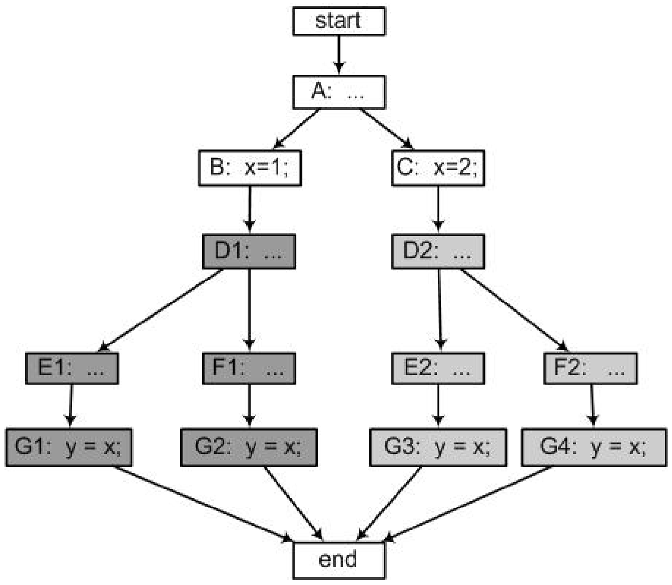


Figure 3.4: Naïve restructuring of the acyclic program $P1$ of Figure 3.1 No destructive merges. *Use* of variable x at nodes $G1$, $G2$, $G3$ and $G4$ can be replaced by constants.

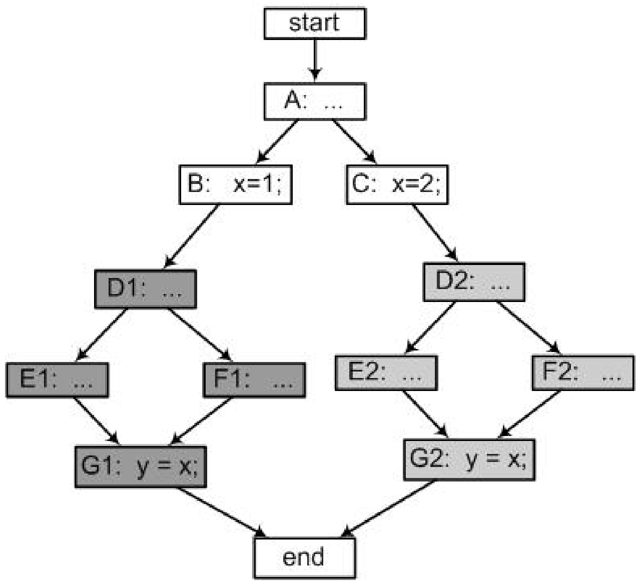


Figure 3.5: Split Graph $S1$ corresponding to the acyclic program of Figure 3.1 constructed by our approach. No destructive merges. *Use* of variable x at nodes $G1$ and $G2$ can be replaced by constants.

sary. In our specific example, only two copies of node G are sufficient (and necessary) as seen in Figure 3.5. The *Naïve* approach has four copies of node G. To further illustrate the code size increase, consider the program template in Figure 3.6 which is a generalisation of the program of Figure 3.1. The program consists of n control-flow merges $D1$, $D2$,..,Dn. The *Naïve* approach would transform the program of Figure 3.6 having $\theta(n)$ nodes to one having $\theta(2^n)$ nodes as seen in Figure 3.7. Our approach would create a transformed CFG having $\theta(2n)$ nodes as shown in Figure 3.8.

Thus, we see that such a *Naïve* approach fails to be efficient and leads to unnecessary increase in code size. Of course, there are instances of programs for which our approach and the *Naïve* approach would construct the same restructured program. But, in general, our approach is better than this *Naïve* approach.

3.1.2 Computing The Region Of Influence

Single Destructive Merge

In this section, we look at which nodes need to be duplicated in order to improve data-flow precision, given a destructive merge. Consider program P2

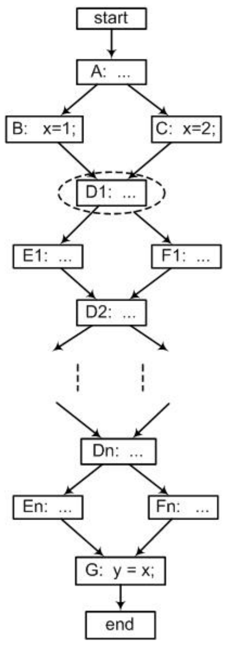


Figure 3.6: Generalisation of acyclic program $P1$ of Figure 3.1 consisting of n control-flow merges. Node $D1$ is a destructive merge. *Use* of variable x at nodes Dn cannot be replaced by a constant.

in Figure 3.9, which is a slight variation of the program $P1$ in Figure 3.1. The transformed graph $S2$ corresponding to program $P2$ is shown in Figure 3.10. We will begin by examining programs $P2$ and $S2$ in greater detail in order to explain how we obtained the Split Graph.

In program $P2$, node D is a destructive merge with data-flow facts $\{\mathtt{x} = \mathtt{2}\}$ and $\{\mathtt{x} = \mathtt{1}\}$ merging to obtain the data-flow fact $\{\mathtt{x} = \mathtt{nc}\}$. The first point to note is that if (somehow) the data-flow fact $\{\mathtt{x} = \mathtt{1}\}$ were to hold at node D, then the *use* of variable x at node G can be replaced by the constant 1. Thus, we see that in order to optimize node G, it is *useful* for data-flow fact $\{\mathtt{x} = \mathtt{1}\}$ to hold at node D. This also holds for the data-flow fact $\{\mathtt{x} = \mathtt{2}\}$. On the other hand, it is not useful for the data-flow fact $\{\mathtt{x} = \mathtt{nc}\}$ to hold at node D. Hence, in program $S2$ in Figure 3.10 we see that at the copies $D1$ and $D2$ of D, the data-flow facts $\{\mathtt{x} = \mathtt{1}\}$ and $\{\mathtt{x} = \mathtt{2}\}$ hold respectively. Further, nodes H, I and J cannot be optimized even if the data-flow facts $\{\mathtt{x} = \mathtt{1}\}$ or $\{\mathtt{x} = \mathtt{2}\}$ hold at node D. Thus, these nodes are not duplicated in Program $S2$ in Figure 3.10. We say that these nodes are not *influenced* by the destructive merge at node D. The nodes D, E, F, and G are called the *Region of Influence* and are exactly those nodes which have multiple copies in program $S2$.

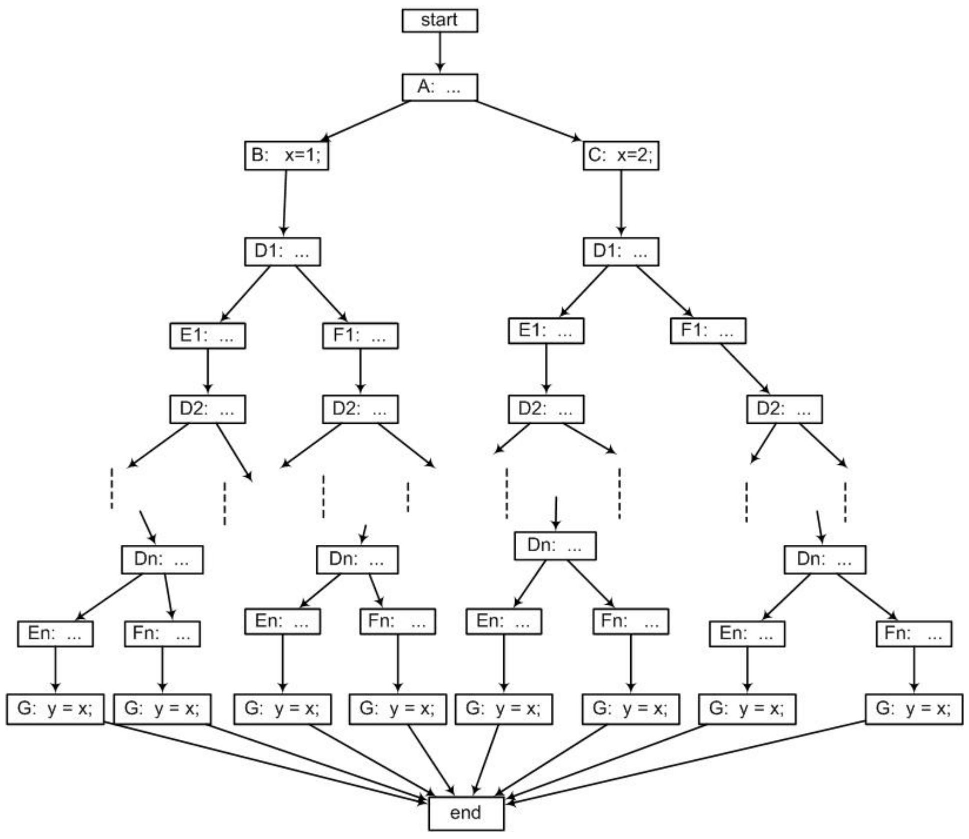


Figure 3.7: The exponential blowup of code as seen when the Naïveapproach is applied to the acylic program of Figure 3.1.

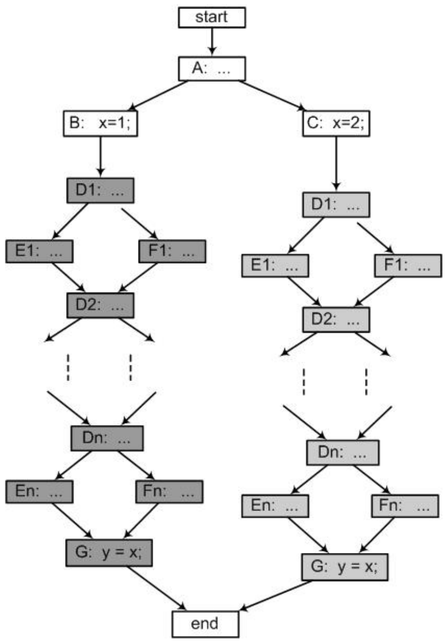


Figure 3.8: Result of applying the Split Approach to the acylic program of Figure 3.1.

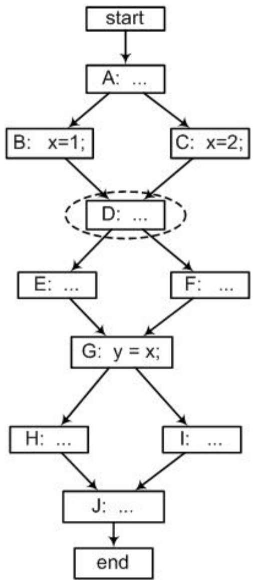


Figure 3.9: Simple acyclic program $P2$. Node D is a destructive merge. *Use* of variable x cannot be replaced by a constant.

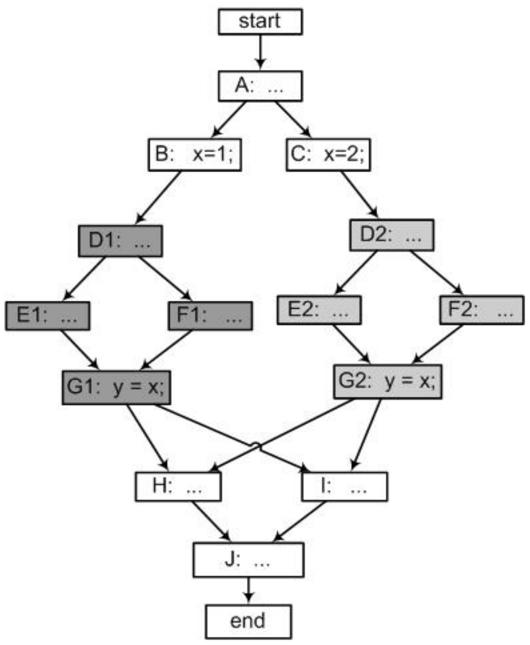


Figure 3.10: Split Graph $S2$ corresponding to the acyclic program of Figure 3.9 constructed by our approach. No destructive merges. *Use* of variable x at nodes $G1$ and $G2$ can be replaced by constants.

Next, we formalise these concepts for a general forward data-flow problem. The following definitions assume that we are given CFG and a data-flow problem and a corresponding solution.

Definition 24. (Useful Data-flow Facts) *For a given node n, a data-flow fact is said to be a* Useful Data-flow Fact, *i.e. $d \in$* `useful_dff`(n), *iff data-flow fact d holding true at node n implies that the node n can be optimized.*

The above definition implicitly captures the interaction between a data-flow analysis and the compiler optimization i.e. the client. It allows us to abstract out the details of this relation by providing an oracle which knows which data-flow facts enable the optimization for a particular program statement. We can now generalise this notion of useful data-flow facts.

Definition 25. (Useful Data-flow Facts) *Given nodes m and n, a data-flow fact is said to be* Useful Data-flow Fact *for node n at node m, i.e. $d \in$* `useful_dff`(m, n), *iff data-flow fact d holding true at node m implies that the node n can be optimized.*

It is easy to see that Definition 24 is a special case of Definition 25 with node m being the same node n.

Example. Consider the program in Figure 3.9.

$$\texttt{useful\_dff}(G) = \{\ldots, \texttt{x} = -1, \texttt{x} = 0, \texttt{x} = 1, \texttt{x} = 2, \ldots\},$$

$$\texttt{useful\_dff}(D, G) = \{\ldots, \texttt{x} = -1, \texttt{x} = 0, \texttt{x} = 1, \texttt{x} = 2, \ldots\}.$$

This is due to the fact that if the value of variable x were to be a constant at node D, then it would remain a constant at node G. This would enable us to optimize node G by replacing the *use* of variable x with a constant.

$\square$

Definition 26. (Influenced Nodes) *Given a destructive merge m, we say a node n is* influenced *by the destructive merge m, i.e., $n \in \texttt{influenced\_nodes}(m)$, iff*

$$\texttt{destroyed\_dff}(m) \cap \texttt{useful\_dff}(m, n) \neq \emptyset.$$

Definition 27. (Revival Data-flow Facts) *Given a destructive merge m, a data-flow fact is said to be a* Revival Data-flow Fact*, i.e., $d \in \texttt{revival\_dff}(m)$, iff there exists a node $n \in \texttt{influenced\_nodes}(m)$ such that*

$$d \in \texttt{destroyed\_dff}(m) \cap \texttt{useful\_dff}(m, n).$$

In other words, a node n is *influenced* by a destructive merge m when, if one of the data-flow facts d which is destroyed by the node m were to hold true at node m, then node n could be optimized. Intuitively, a node is *influenced* by a destructive merge if eliminating the destructive merge can enable the optimization of the node. Further, these are the only nodes which can be optimized if the destructive merge is eliminated. The Revival data-flow facts denote those data-flow facts which if they would hold true at the destructive merge m would enable the optimization of some influenced node.

Example. Consider Figure 3.9,

$$\texttt{influenced\_nodes}(m) = \{G\},$$

$$\texttt{destroyed\_dff}(m) = \{\texttt{x} = 1, \texttt{x} = 2\},$$

$$\texttt{useful\_dff}(D, G) = \{\ldots, \texttt{x} = -1, \texttt{x} = 0, \texttt{x} = 1, \texttt{x} = 2, \ldots\},$$

$$\texttt{revival\_dff}(m) = \{\texttt{x} = 1, \texttt{x} = 2\}.$$

$\square$

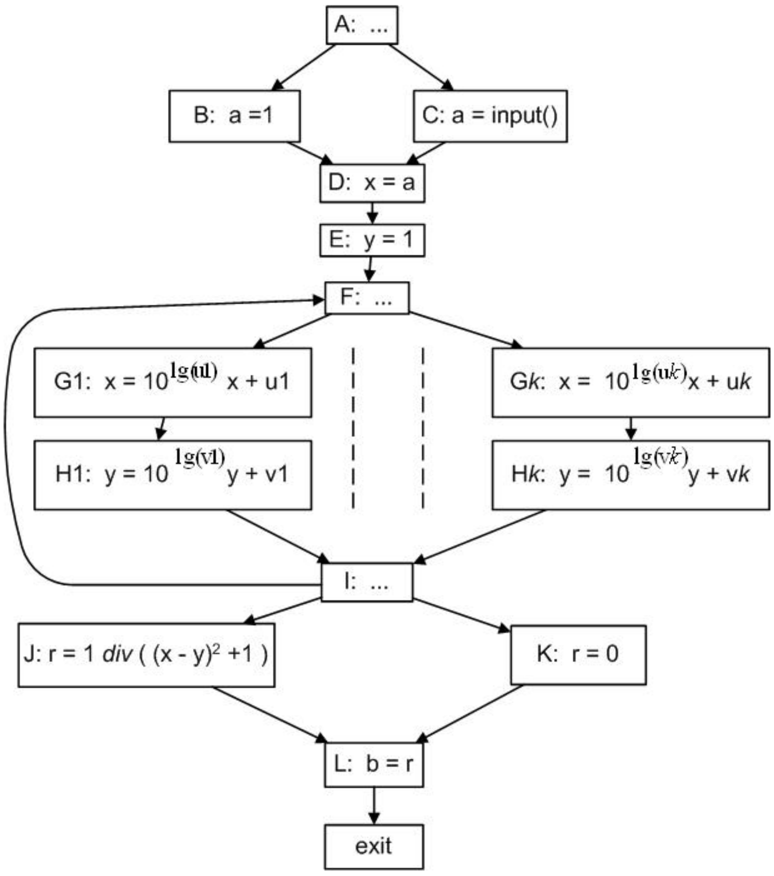


Figure 3.11: Schematic diagram illustrating reduction of Post Correspondence problem for Theoreom 1.

Since the influenced nodes are the only nodes which can be optimized by eliminating the destructive merge, we would like to determine the largest such set of nodes. However, for the specific problem of Constant Propagation we establish that determining whether a node n belongs to `influenced_nodes`(m) for a destructive merge m is undecidable in general.

Theorem 1. (Influence Theorem) *For the problem of Constant Propagation, given a node n and destructive merge m determining if $n \in$ `influenced_nodes`(m) is undecidable.*

Proof. The proof follows from the undecidability of Constant Propagation for programs with loops even if all branches are considered to be non-deterministic as described in [16], and originally stated by Hecht [11] and by Reif and Lewis [20]. The proof is based on the reduction of the Post correspondence problem [13]. We show that a node $n \in$ `influenced_nodes`(m) iff the Post correspondence problem is not solvable.

A Post correspodence system consists of pairs of strings $(u_1, v_1), \ldots, (u_k, v_k)$ with $(u_i, v_i) \in \{0, 1\}^*$. The Post correspondence problem (PCP) consists of determining whether there exists a sequence $i_1, i_2, \ldots, i_n, 1 \leq i_j \leq k$, such that $u_{i_1} \cdot \ldots \cdot u_{i_n} = v_{i_1} \cdot \ldots \cdot v_{i_n}$, where $\cdot$ represents string concatenation. It is well-known that the PCP is undecidable [13].

Figure 3.11 illustrates the reduction. The variable x and y represent the decimal representation of strings over $\{0, 1\}^*$. Node D is a destructive merge with `destroyed_dff`$(D) = \{\mathtt{a} = \mathtt{1}\}$. Further, for the assignment statement $b = r$ at node L, `useful_dff`$(L) = \{\ldots, \mathtt{r} = -\mathtt{1}, \mathtt{r} = \mathtt{0}, \mathtt{r} = \mathtt{1}, \mathtt{r} = \mathtt{2}, \ldots\}$. Using Definition 26, node $L \in$ `influenced_nodes`(D) iff `destroyed_dff`$(D) \cap$ `useful_dff`$(D, L) \neq \emptyset$. Thus, we would like to know whether the data-flow fact $\{\mathtt{a} = \mathtt{1}\}$ belongs to the set `useful_dff`(D, L). In other words, to determine whether node $L \in$ `influenced_nodes`(D) we have to show that variable r has a constant value at node L if the data-flow fact $\{\mathtt{a} = \mathtt{1}\}$ holds true at node D.

In Figure 3.11, for each pair of the correspondence system, there is a branch of the loop which appends strings u_i and v_i to x and y, respectively. This is achieved by left shifting the digits of x and y by $lg(u_i)$ and $lg(v_i)$ digits first, where $lg(u_i)$ and $lg(v_i)$ are the lengths of the decimel representation of u_i and v_i respectively. Then u_i and v_i are added.

At the exit of the loop, the value of the expression $(x - y)$ will *always* be non-zero if the Post correspondence problem has no solution. In this case the expression 1 `div` $((x - y)^2 + 1)$ *always* evaluates to 0, where `div` represents integer division. On the other hand, if the Post correspondence is solvable, this expression *can* evaluate to 1. Thus, r is a constant with value 0 at node L iff the Post Correspondence problem is not solvable. The reason for the assignment $r = 0$ at node K is to exclude the case in which r takes the constant value 1, which occurs when the Post correspondence system is universally solvable.

Thus, we have shown that if the dataflow fact $\{\mathtt{a} = \mathtt{1}\}$ holds at node D then variable r is a constant at node L iff the Post correspondence problem is not solvable. It follows that, in general, determining whether a node $n \in$ `influenced_nodes`(m), where m is a destructive merge, is undecidable for the problem of Constant Propagation.

□

As we see in the Section 3.1.3, our restructuring is guaranteed to improve data-flow precision and optimize the nodes which are *influenced* by the destructive merge. The above theorem implies that we cannot determine all

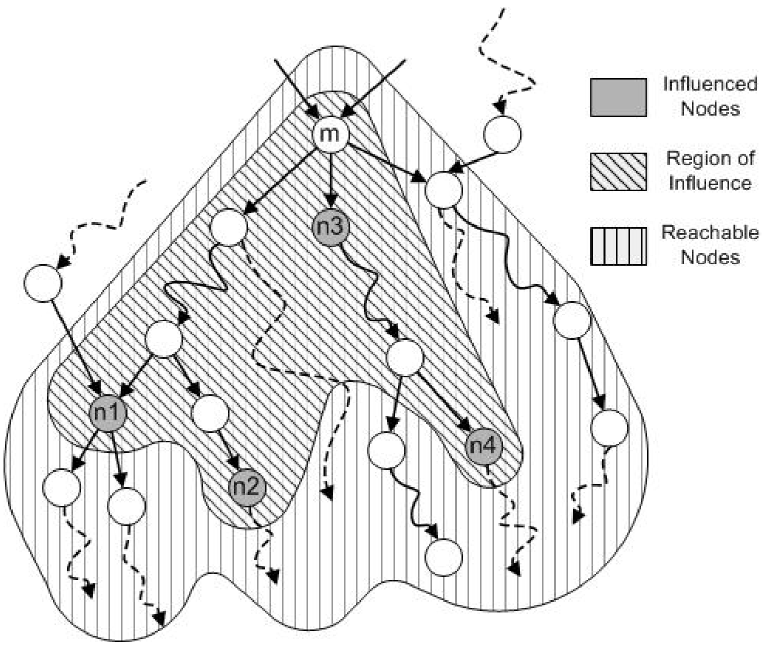


Figure 3.12: Schematic diagram showing the Region of Influence for a destructive merge m.

nodes which are *influenced* by a destructive merge. If we include a node which is not actually influenced by the destructive merge in `influenced_nodes`(m), then the restructuring will not enable any optimizations. Thus, the corresponding increase in code size will be unnecessary. Hence, in practice, we under-approximate the set of *influenced* nodes. In particular, for Constant Propagation, if the data-flow fact for variable x is destroyed at merge node m, then the *uses* that are reachable only along paths from node m that do not contain any definitions of variable x are *influenced* by the destructive merge m. This set is an under-approximation and there might be nodes which are influenced by the destructive merge m and not be included in the set `influenced_nodes`(m) which is actually determined.

Definition 28. (Region of Influence) *Given a destructive merge m, a node n is said be in the* Region of Influence, *i.e., $n \in$* `RoI`(m), *iff $n \in$* `reachable_nodes`(m) *and there exists a node $u \in$* `influence_nodes`(m) *and $u \in$* `reachable_nodes`(n).

Figure 3.12 shows a schematic diagram showing `RoI`(m) for the destructive merge m.

Having determined the set of influenced nodes, the Region of Influence consists of those nodes which are sufficient and necessary to be duplicated in order to improve precision and optimize the influenced nodes.

Example. In Figure 3.9, node D is a destructive merge.

$$\texttt{influenced\_nodes}(D) = \{G\},$$

$$\texttt{reachable\_nodes}D = \{D, E, F, G, H, I, J\},$$

$$G \in \texttt{reachable\_nodes}(D),\ G \in \texttt{reachable\_nodes}(E),$$

$$G \in \texttt{reachable\_nodes}(F),\ G \in \texttt{reachable\_nodes}(G),$$

$$\texttt{RoI}(m) = \{D, E, F, G\}.$$

□

Multiple Destructive Merges

In the previous section, we dealt with the situation where we were given a single destructive merge to eliminate. In practice, we will have a set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$ which are to be eliminated. As mentioned earlier, the method to determine this set will be given in Chapter 4. In this section, we extend the concepts of the previous section to handle multiple destructive merges viz. we determine which nodes are to be duplicated in order to improve data-flow precision.

Definition 29. (Influenced Nodes) *Given a set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$, the set of* Influenced Nodes *is defined as*

$$\texttt{influenced\_nodes}(\mathcal{M}) = \bigcup_{m \in \mathcal{M}} \texttt{influenced\_nodes}(m)$$

Definition 30. (Region of Influence) *Given a set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$, the* Region of Influence *corresponding to $\mathcal{M}$ is defined as*

$$\texttt{RoI}(\mathcal{M}) = \bigcup_{m \in \mathcal{M}} \texttt{RoI}(m)$$

Extending the definitions to handle multiple destructive merges is straight forward. A node n belongs to $\texttt{influenced\_nodes}(\mathcal{M})$ iff it is *influenced* by at least one destructive merge $m \in \mathcal{M}$. Similarly, a node n belongs to $\texttt{RoI}(\mathcal{M})$ iff it belongs to the *Region of Influence* of at least one destructive merge $m \in \mathcal{M}$.

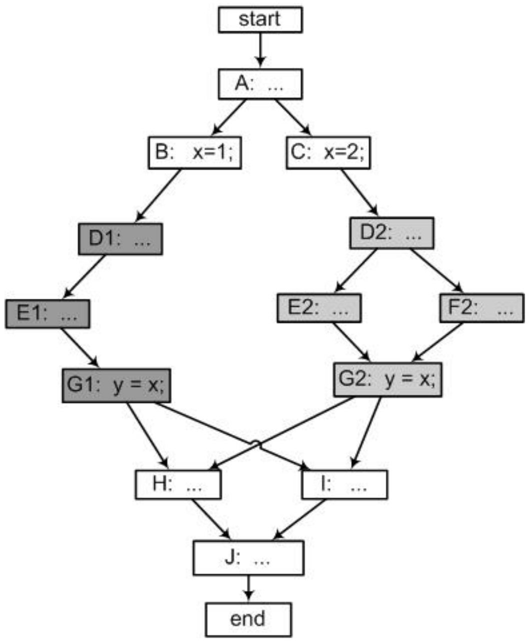


Figure 3.13: Program $W1$ which violates the *Equivalence* constraint, but not the *Efficacy* constraint. The outgoing edges of node $D1$ are incorrect.

3.1.3 The CFG Restructuring

We continue with our example program $P2$ shown in Figure 3.9. Having identified which data-flow facts should hold at node D in order to optimize node G viz., useful_dff(D, G), and which nodes should be duplicated viz. RoI(D), we will explain the actual control-flow transformation.

There are two main constraints such a transformation should satisfy.

- **Equivalence.** The original and transformed programs should be equivalent.
- **Efficacy.** The transformation should guarantee that the improvement in data-flow precision leads to optimization opportunities.

The program $W1$ of Figure 3.13 illustrates an incorrectly transformed program which violates the *equivalence* constraint. This is because the path $B \rightarrow\ D \rightarrow\ F \rightarrow\ G \rightarrow$ present in Program $P2$ shown in Figure 3.9 is not present in Program $W1$. In Program $P2$, node D has outgoing edges to nodes E and F, while node $D1$ in Program $W1$ has only one outgoing edge to node $E1$.

Program $W2$ in Figure 3.14 satisfies the *equivalence* constraint but violates the *efficacy* constraint. This transformation does not expose new opti-

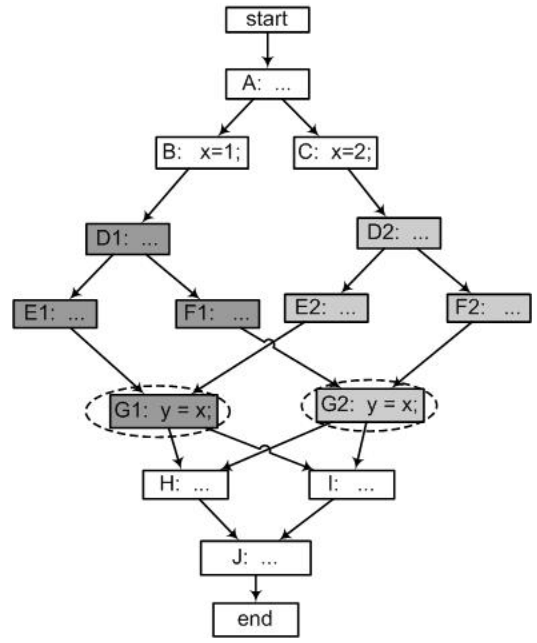


Figure 3.14: Program $W2$ which violates the *Efficacy* constraint, but not the *Equivalence* constraint. Nodes $G1$ and $G2$ are destructive merges and the *use* of variable x cannot be replaced by a constant.

mization opportunities, since nodes G1 and G2 are destructive merges and the *uses* of variable x cannot be replaced by constants. But the *equivalence* constraint is not violated.

Definition 31. (Kill Edges) *Given a Region of Influence for a destructive merge m, an edge $e = (u, v)$ is a* Kill Edge*, i.e. $e \in \texttt{kill\_edges}(m)$, iff $u \in \texttt{RoI}(m)$ and $v \notin \texttt{RoI}(m)$.*

In other words, Kill Edges are those edges whose source node is in the Region of Influence and target node is not in the Region of Influence for a destructive merge m.

Definition 32. (Revival Edges) *Given a Region of Influence for a destructive merge m, an edge $e = (u, m)$ is said to be a* Revival Edge*, i.e. $e \in \texttt{revival\_edges}(m)$, iff*

$$\texttt{out\_dff}(u) \in \texttt{revival\_dff}(m).$$

In other words, Revival Edges are those incoming edges of the destructive merge which correspond to Revival Data-flow facts. Further, let $d_1, d_2, \ldots d_k$ be the k *distinct* Revival Data-flow facts for destructive merge m. We can

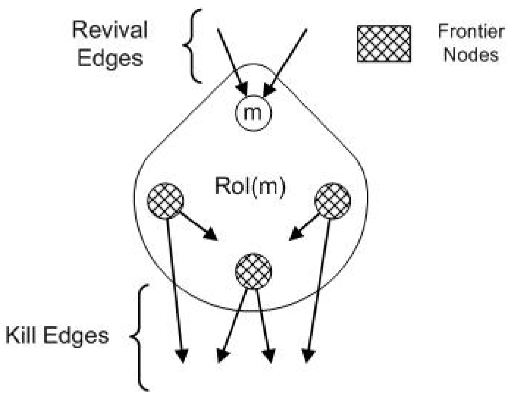


Figure 3.15: Revival and Kill Edges for a Region of Influence corresponding to a destructive merge m.

partition the incoming edges of destructive merge m into $k+1$ equivalence classes $R_0, R_1, \ldots, R_k$. An edge $(u,m) \in R_i, 1 \le i \le k$ if $\texttt{out\_dff}(u) = d_i$, and $(u,m) \in R_0$ otherwise.

Figure 3.15 shows a schematic diagram illustrating the above concepts.

Example. In Figure 3.9,

$$\texttt{out\_dff}(B) = \{\texttt{x} = 1\},\ \texttt{out\_dff}(C) = \{\texttt{x} = 2\},$$

$$\texttt{revival\_dff}(m) = \{\texttt{x} = 1, \texttt{x} = 2\},$$

$$\texttt{revival\_edges}(m) = \{(B,D),(C,D)\},$$

$$\texttt{RoI}(m) = \{D,E,F,G\},$$

$$\texttt{kill\_edges}(m) = \{(G,H),(G,I)\}.$$

□

Lemma 2. (Post-dominance Lemma) *Given a destructive merge m, the set of edges* `kill_edges`*(m)* post-dominate *the node m.*

Proof. The notion of generalized post-dominator set can be found in Definition 5.

We claim that all paths p from the node m to the *end* node can be written as $q \cdot (u,v) \cdot r$ where

1. subpaths q, r may be empty,
2. $\exists w \in \texttt{influenced\_nodes}(m)$ s.t. $w \in \texttt{reachable\_nodes}(u)$,

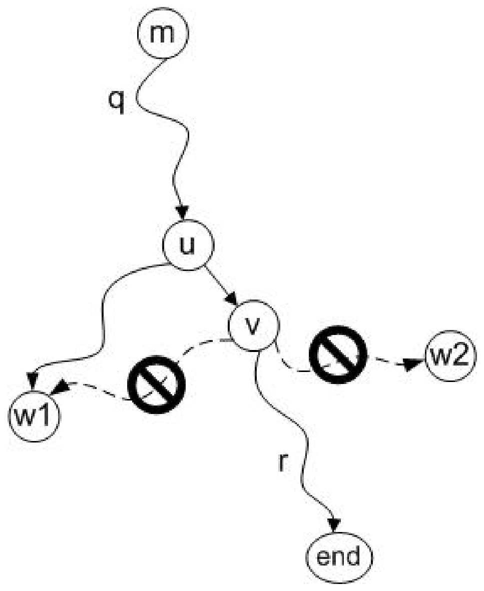


Figure 3.16: Illustrating a path $p = q \cdot (u, v) \cdot r$ from destructive merge m to the *end* node for Lemma 2. $w1, w2 \in$ `influenced_nodes`(m). $w1 \in$ `reachable_nodes`(u). $w1, w2 \notin$ `reachable_nodes`(v).

3. $\nexists w \in$ `influenced_nodes`(m) s.t. $w \in$ `reachable_nodes`(v),

4. subpath r does not contain any node $w \in$ `influenced_nodes`(m)

If we prove this claim, then it is easy to see from points 2 and 3 above that the edge (u, v) is a Kill Edge, and it follows that all paths from destructive merge m to the *end* node pass through a Kill Edge. We now prove the above claim.

We define u to be the *last* node along path p such that $\exists w.w \in$ `influenced_nodes`$(m) \wedge w \in$ `reachable_nodes`(u). Subpath q is the prefix of path p before this node u. Since all influenced nodes are reachable from the destructive merge m such a node u always exists. If node m is chosen as u then subpath q is empty. Having found node u, node v is simply the successor of node u along the path p. Node v may be the *end* node, in which case the subpath r is empty.

Our choice of node u also implies that the subpath r of path p does not contain any node $w \in$ `influenced_nodes`(m)[1]. This proves the claim.

□

Armed with the above concepts we are now ready to define the Split Automaton.

[1] Recall $\forall u, u \in$ `reachable_nodes`(u).

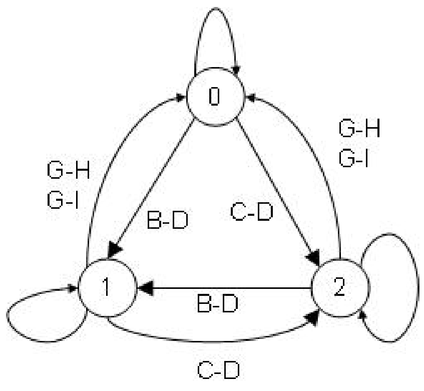


Figure 3.17: The Split Automaton A_D corresponding to destructive merge D in Figure 3.9.

Definition 33. (Split Automaton) *The* Split Automaton A_m *corresponding to a destructive merge m is a finite-state automaton defined as:*

- *the input alphabet $\Sigma = E$, the set of all edges in the CFG,*
- *a set of $k+1$ states $Q = \{s_0, s_1, s_2, \ldots, s_k\}$,*
- *$s_0 \in Q$ is the initial and accepting state,*
- *the transition function $\delta : Q \times \Sigma \rightarrow Q$ defined as*
  $(s_i, e) \rightarrow s_j, e \in R_j$ (Revival Transitions)
  $(s_i, e) \rightarrow s_0, e \in \texttt{kill\_edges}(m)$ (Kill Transitions)
  $(s_i, e) \rightarrow s_i, otherwise$ (No Transition)

Intuitively, state $s_i, 1 \leq i \leq k$ corresponds to Revival Data-flow fact d_i and whenever an edge $e \in R_i$ is seen, the Split Automaton makes a transition to state s_i. We call this the Revival Transition. Further, whenever a Kill edge is seen, the Split Automaton transitions to state s_0. We call these the Kill Transitions. In all other cases, the automaton remains in the same state, and makes no transitions.

Example. Figure 3.17 shows the Split Automaton A_D corresponding to the destructive merge D in CFG P in Figure 3.9. Intuitively, whenever the automaton is in state 1, data-flow fact $\{\texttt{x} = 1\}$ holds, while in state 2 data-flow fact $\{\texttt{x} = 2\}$ holds. Further, a transition to state 0 from states 1 or 2 implies that the automaton has stopped duplicating nodes in the CFG.

To get further intuition into the structure of the Spit Automaton, we consider the set of paths from the *start* state upto node G along which data-flow facts $\{\texttt{x} = 1\}$ and $\{\texttt{x} = 2\}$ flow.

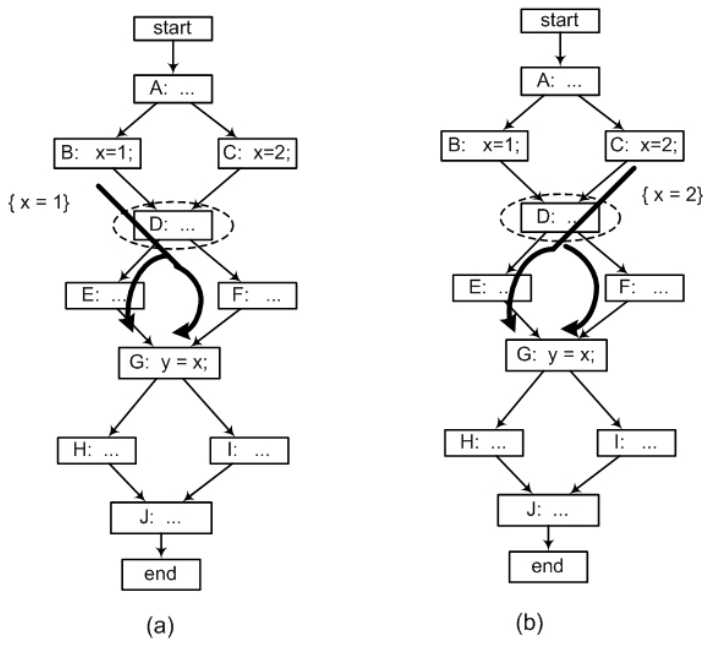


Figure 3.18: *(a)* The two paths along which the data-flow fact $\{\mathtt{x} = \mathtt{1}\}$ reaches the node G. *(b)* The two paths along which the data-flow fact $\{\mathtt{x} = \mathtt{2}\}$ reaches the node G.

Figure 3.18(a) shows the paths along which the data-flow fact $\{\mathtt{x} = \mathtt{1}\}$ holds. Notice that these paths pass through the edge (B, D). In fact, these are exactly those paths which are accepted by the state 1 of the Split Automaton shown in Figure 3.17. Similarly, Figure 3.18(b) shows the paths along which the data-flow fact $\{\mathtt{x} = \mathtt{1}\}$ holds. In this case, these paths pass through the edge $(C, D)$and are exactly those paths which are accepted by the state 2 of the Split Automaton.

Furthermore, if we examine the corresponding Split Graph which we construct shown in Figure 3.10, we see that these two sets of paths never intersect. Thus, the differing data-flow facts along these paths never merge and we don't lose information.

Now, in the Split Automaton, there are transitions back to state 0 on edges (G, H) and (G, I). This is because after node G it is no longer beneficial to separate the paths and we don't lose useful information when the data-flow facts along paths merge.

□

As we have seen already, a CFG can be viewed as a finite-state automaton with nodes of the CFG corresponding to the state of the automaton, and the

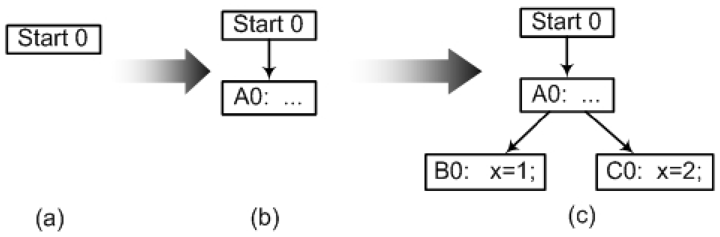


Figure 3.19:

edges defining the alphabet as well as the transition function. The entry and exit node of the CFG correspond to the start and accepting states in the automaton respectively. We call this a *control-flow automaton* (Definition 22).

Definition 34. (SPLIT GRAPH) *Given a CFG P and and a Split Automaton A_m, we define the* Split Graph *S to be $S = P \times A_m$, where $\times$ is the product automaton operator [13].*

Each node n in the Region of Influence of node m in P will have multiple copies n_i in the Split Graph S, each corresponding to the state s_i in the Split Automaton. We refer the reader to the algorithm described in Figure 2.4 for product computation.

Example. The Split Graph $S2$ in Figure 3.10 is obtained by performing the product of the CFG P in Figure 3.9 and the Split Automaton A_D in Figure 3.17.

Figures 3.19–3.21 show the Split Graph being constructed during the product computation. Figure 3.19(a) shows the *start* state of the product automaton which is also the *start* state of the Split Graph. On the edge $(start, A)$ the CFG goes to node A while the automaton stays in state 0. A new state $A0$ is created, and there is a transition from state $start0$ to state $A0$ in the product automaton as seen in Figure 3.19(b). Similarly, for edges (A, B) and (A, C), the automaton stays in state 0 and the CFG transitions to states B and C respectively. The resulting product automaton is shown in Figure 3.19(c).

Now, for edge (B, D), the automaton transitions uses a Revival transition and transitions to state 1. Thus, we see the new state $D1$ and the transition from state $B0$ to $D1$ in the product automaton in Figure 3.20(d). After this the automaton stays in state 1 which gives rise to states $E1$, $F1$, and $G1$. We see the same situation for when the automaton is state $C0$ and transitions on edge (C, D). States $D2$, $E2$, $F2$ and $G2$ are created in the product automaton as shown in Figure 3.20(e).

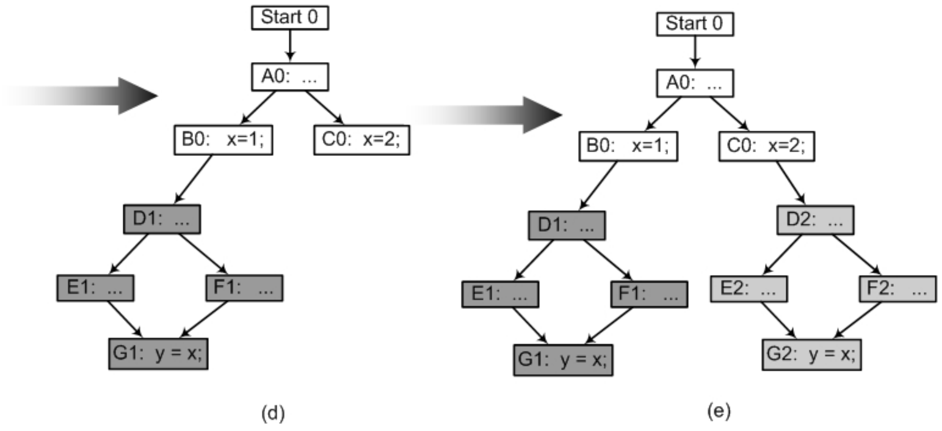


Figure 3.20:

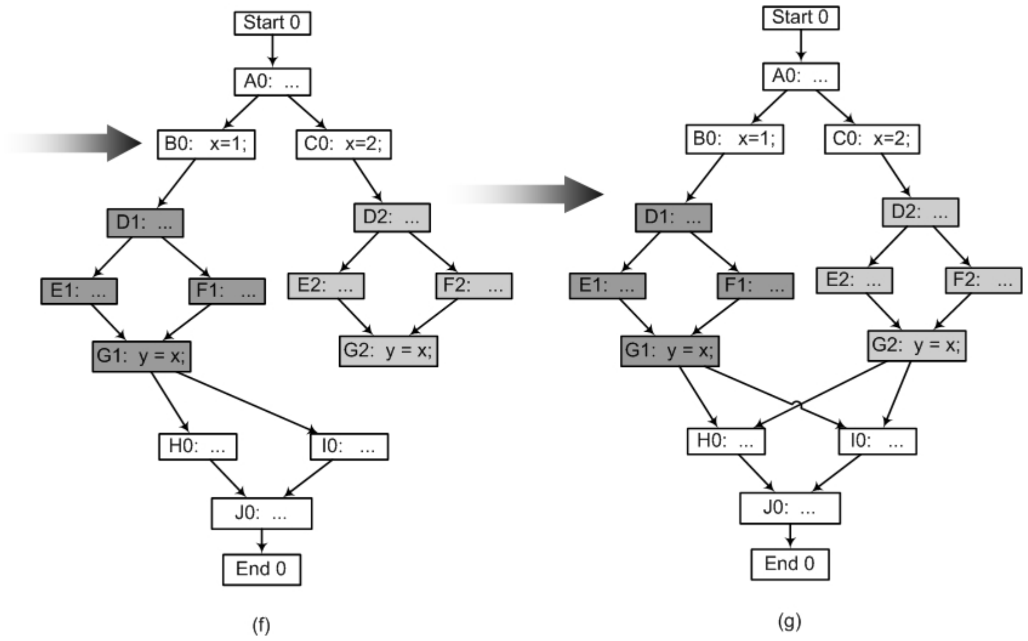


Figure 3.21:

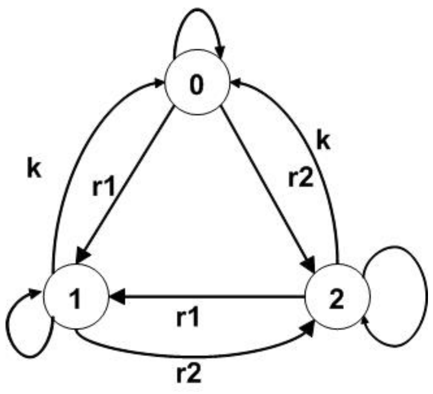


Figure 3.22: The split automaton A_m for destructive merge m. State 0 is the start and accepting state. $r1$ and $r2$ are Revival edges, and k represents the Kill edges.

Consider state $G1$. The split automaton is in state 1. On seeing the edge (G, H), the split automaton transitions back to state 0 which is a Kill transition. Thus, state $H0$ is created and there is a transition from $G1$ to $H0$. Similar, for edge (G, I). After this the automaton stays in state 0 and state $J0$ and $end0$ are created as shown in Figure 3.21(f).

Now consider state $G2$ in the product automaton. The CFG is in node G and the split automaton is in state 2. On seeing the edge (G, I), the split automaton transitions to state 0. Thus, there is a transition from state $G2$ to state $I0$ in the product automaton. But, the state $I0$ already exists in the product automaton and no new node needs to be created. Similarly, for edge (G, H). The final Split Graph is shown in Figure 3.21(g).

□

Lemma 3. (Subsumption Lemma) *Given a CFG P with a destructive merge m and the corresponding split automaton A_m,*
$L(P) \subseteq L(A_m)$.

Proof. For clarity of exposition, we will restrict the split automaton A_m to the form shown in Figure 3.22 i.e. A_m has only three states. The generalization of the proof where A_m has n states is straight forward.

The set of edges E of the CFG P form the alphabet for the automata P and A_m. Hence, it follows that $L(P) \subseteq E^*$ and $L(A_m) \subseteq E^*$. The set E^* can be partitioned into three partitions:

E1: Strings not containing r_1 or r_2.

E2: Strings of the form $q \cdot r_1 \cdot t$, where q can be the empty string and t does not contain r_1 or r_2.

E3: Strings of the form $q \cdot r_2 \cdot t$, where q can be the empty string and t does not contain r_1 or r_2.

It is easy to see that every string in E^* belongs to at least one of these partitions, and any string in E^* belongs to only one such partition. Thus, $E1$, $E2$ and $E3$ partition the set E^* into three equivalence classes.

Consider any string $p \in \mathrm{E1} \cap L(P)$ i.e. p does not contain $r1$ or $r2$, *and* represents a control-flow path beginning at the *start* node and ending at the *end* node of the CFG P. We shall trace this string through the automaton A_m (Figure 3.22) and see whether the string is accepted. Since p contains neither $r1$ nor $r2$, the automaton will never transition out of state 0. State 0 being the accepting state of automaton A_m, string p will be accepted. Thus, it follows that all strings belonging to equivalence class $\mathrm{E1} \cap L(P)$ will be accepted by automaton A_m.

Consider any string $p \in \mathrm{E2} \cap L(P)$ i.e. p is of the form $q \cdot r_1 \cdot t$, where q can be the empty string and t does not contain r_1 or r_2., *and* represents a control-flow path beginning at the *start* node and ending at the *end* node of the CFG P. Again, let us trace this string through the automaton A_m. Suppose that the automaton A_m is in state s after reading the (possibly empty) string q. Now, because of the nature of the transition relation, after reading the subsequent r_1, the automaton A_m will transition to state 1. Since the remaining part of the string p, viz. string t, contains neither $r1$ nor $r2$, the automaton A_m will remain in state 1 unless it sees an alphabet from the kill set k.

By the Post-dominance Lemma (Lemma 2), the kill edges k post-dominate the destructive merge m. Thus, the string t is of the form $t_1 \cdot g \cdot t_2$, where t_1 and t_2 may be empty, $g \in k$, and t_2 does not contain any alphabet (edge) from k. Thus, the automaton A_m has to make a transition from state 1 to state 0 after reading g. In state 0 the remaining part of the string i.e. t_2 will not cause any transitions and the automaton will remain in state 0. Thus, the string $p \in \mathrm{E2} \cap L(P)$ is accepted by automaton A_m.

Similarly, we can show that any string $p \in \mathrm{E3} \cap L(P)$ is accepted by automaton A_m.

Hence, we see that $L(P) \subseteq L(A_m)$.

□

Intuitively, the Subsumption Lemma implies that A_m is a control-flow abstraction of P and accepts more words.

Theorem 2. (Equivalence Theorem) *Consider a CFG P with a destructive merge m. If the corresponding split automaton is A_m, the the Split Graph $S = P \times A_m$ is* equivalent *to the original CFG P i.e. $S \equiv P$.*

Proof. Since S is the product of P and A_m, by the Intersection Lemma (Lemma 1), we can say that

$$L(S) = L(P) \cap L(A_m) \tag{3.1}$$

Further, by the Subsumption Lemma (Lemma 3),

$$L(P) \subseteq L(A_m) \tag{3.2}$$

Using Equation 3.2 in Equation 3.1, we can infer that

$$L(S) = L(P) \tag{3.3}$$

Thus, from Equation 3.3 and Definition 23,

$$S \equiv P.$$

□

Thus, using simple concepts from automaton theory [13], we are able to prove the correctness of our restructuring transformation. Next, we prove the efficacy of the restructuring.

Theorem 3. (Efficacy Theorem) *If node $n \in$ `influenced_nodes`(m) in CFG P, then in the Split Graph S, node $n_i, i \neq 0$ can be optimized, where $S = P \times A_m$.*

Proof. As before, for clarity of exposition, we will restrict the split automaton A_m to the form shown in Figure 3.22 i.e. A_m has only three states. The generalization of the proof where A_m has n states is straight forward.

Consider Figure 3.23(a). Node m is a destructive merge and node $n \in$ `influenced_nodes`(m). Let d_1 be a Revival Data-flow fact and r_1 be the corresponding equivalence class of edges. Data-flow fact d_1 holds at the *out* of node u. Since $n \in$ `influenced_nodes`(m), by definition, if d_1 (somehow) holds true at node m, then node n can be optimized in CFG P. In other words, if $d1$ holds at *in* of node m, then some $d_3 \in$ `useful_dff`(n) will hold at *in* of node n. Here, data-flow fact $d3$ will hold at *in* of node n when considering all paths such as p_1, p_2, and p_3.

The theorem proceeds in two steps. We first show that data-flow fact d_1 holds true at the *in* of node m_1 in the Split Graph S. Then, we show that this data-flow fact does not merge with other differing data-flow facts.

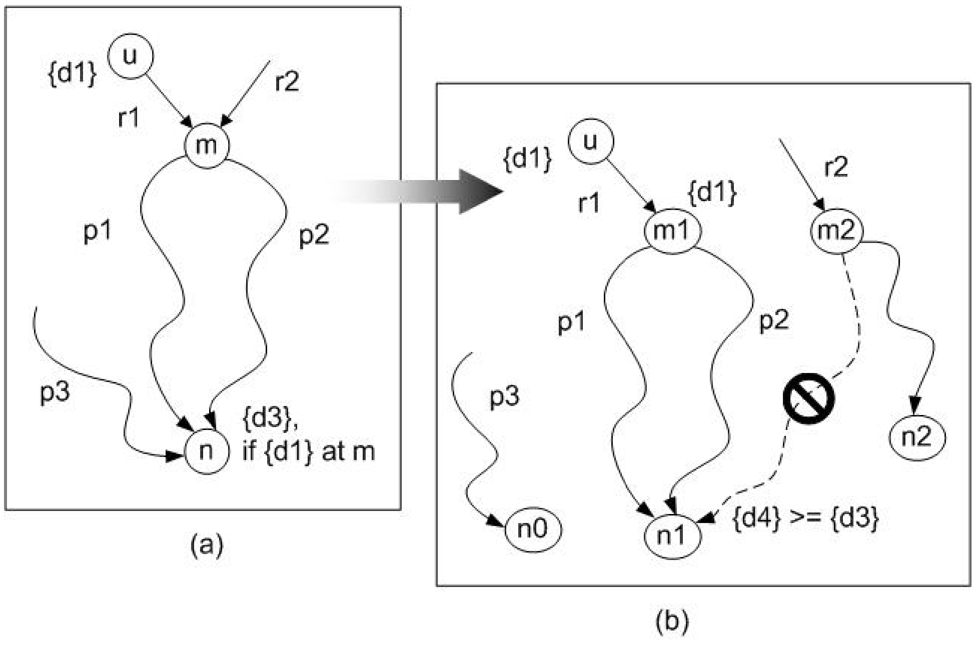


Figure 3.23: Refer to text of Theorem 3 for details. *(a)* Split Graph S. Paths $p1$, $p2$, and $p3$ represent the various paths which reach the node $n \in$ `influenced_nodes`(m) *(b)* Program P *after* restructuring. The only paths which reach $n1$ are $p1$ and $p2$ which pass through node $m1$.

All paths reaching nodes with suffix 1 in the split graph S are of the form $q \cdot (u, m) \cdot r$, where subpaths q and r may be empty, and r does not contain any revival edges. In particular, all paths reaching node $m1$ in split graph S are of the form $q \cdot (u, m)$. Since data-flow fact d_1 holds true at node u in program P, it holds at node u in split graph S. Further, since node $m1$ is the only successor of node u in split graph S, data-flow fact d_1 holds true at *in* of node $m1$. This is illustrated in Figure 3.23(b). This proves the first part of our proof.

Further, all paths reaching node $n1$ are also of the form $q \cdot (u, m) \cdot r$ mentioned above. Thus, data-flow information along paths such as $p3$ which do not pass through node m, or along paths which pass through other revival edges do not reach node $n1$. Thus, at the *in* of node $n1$ some $d4 \leq d3$ will hold. This proves the second part of our proof.

Thus, it follows that data-flow fact d_1 holds at *in* of node $m1$ in split graph S and some $d_3 \in$ `useful_dff`(n) holds at *in* of node $n1$. Thus, node $n1$ can be optimized in the split graph.

□

Multiple Destructive Merges

We now consider eliminating multiple destructive merges. Consider the set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$ which are to be eliminated. Let $A_1, A_2, \ldots, A_k$ be the corresponding Split Automata.

Definition 35. (SPLIT GRAPH) *Given a CFG P and and a set of Split Automata $A_1, A_2, \ldots, A_k$, we define the* Split Graph *S to be $S = P \times A_1 \times A_2 \times \ldots \times A_k$, where $\times$ is the product automaton operator [13].*

In this general setting, a node n in the CFG P can have *at most* $c_1 * c_2 * \ldots * c_k$ copies in the Split Graph S. We represent each node in the Split Graph using the label of the node in the original graph and a *vector* of suffixes of length k i.e. the number of split automata. We use the notation n_i^j to represent those copies of the node n in the Split Graph which have the jth component of the vector set to i.

The correctness and efficacy results also hold in this general setting. It is interesting to note that we placed no restriction of the nature of the CFG. Thus, our restructuring can handle programs with complex loop structures.

Theorem 4. (EQUIVALENCE THEOREM FOR MULTIPLE DESTRUCTIVE MERGES) *Consider a CFG P with destructive merges $m_1, m_2, \ldots, m_k$. If the corresponding split automata are $A_1, A_2, \ldots, A_k$, then the Split Graph $S_k = P \times A_1 \times A_2 \times \ldots \times A_k$ is* equivalent *to the original CFG P i.e. $S_k \equiv P$.*

Proof. We prove this theorem by Induction on the number of destructive merges.

The *Base* case is when there is a single destructive merge. i.e.

$$S_1 = P \times A_1.$$

By Theorem 2, we have

$$S_1 \equiv P.$$

Now for the *Induction* step. Let us assume that the theorem is true for $j < k$ destructive merges. Thus, we have

$$S_j = P \times A_1 \times A_2 \times \ldots \times A_j, \; j < k \tag{3.4}$$

and

$$S_j \equiv P. \tag{3.5}$$

Now consider

$$S_{j+1} = P \times A_1 \times A_2 \times \ldots \times A_j \times A_{j+1}, \; j \leq k.$$

Substituting Equation 3.4,

$$S_{j+1} = S_j \times A_{j+1}$$

By the Intersection Lemma (Lemma 1),

$$L(S_{j+1}) = L(S_j) \cap L(A_{j+1})$$

Using Equation 3.5,

$$L(S_{j+1}) = L(P) \cap L(A_{j+1}) \tag{3.6}$$

Using the Subsumption Lemma (Lemma 3), we have

$$L(P) \subseteq L(A_{j+1}) \tag{3.7}$$

Using Equation 3.6 and 3.7,

$$L(S_{j+1}) = L(P)$$

Thus, we have

$$S_{j+1} \equiv P$$

Thus, by induction,

$$S_k \equiv P$$

□

Lemma 4. (Monotonicity Lemma) *Given a node n in the CFG P and any corresponding node n_q in the Split Graph S, $\texttt{in\_dff}(n) \preceq \texttt{in\_dff}(n_q)$, where $S = P \times A_1 \times A_2 \times \ldots \times A_k$.*

Proof. Let P be the set of paths from the entry node to node n and $\texttt{dff}(p)$, $p \in P$ be the data-flow fact from path p.

By definition,

$$\texttt{in\_dff}(n) = \bigwedge_{p \in P} \texttt{dff}(p) \tag{3.8}$$

Let Q be the set of states of the product automaton $A = A_1 \times A_2 \times \ldots \times A_k$ and $P_q \subseteq P$ be the set of paths which drive A to the state $q \in Q$.

Thus, we have

$$\bigwedge_{p \in P} \texttt{dff}(p) = \bigwedge_{q \in Q} \bigwedge_{p \in P_q} \texttt{dff}(p) \tag{3.9}$$

Removing the outer meet from the right hand side of Equation 3.9

$$\bigwedge_{p \in P} \texttt{dff}(p) \preceq \bigwedge_{p \in P_q} \texttt{dff}(p), \forall q \in Q \tag{3.10}$$

Substituting Equation 3.10 in Equation 3.8 we get,

$$\texttt{in\_dff}(n) \preceq \bigwedge_{p \in P_q} \texttt{dff}(p), \forall q \in Q \tag{3.11}$$

Further in the Split graph, by definition,

$$\texttt{in\_dff}(n_q) = \bigwedge_{p \in P_q} \texttt{dff}(p) \tag{3.12}$$

Using Equation 3.12 in Equation 3.11, we get

$$\texttt{in\_dff}(n) \preceq \texttt{in\_dff}(n_q), \forall q \in Q \tag{3.13}$$

□

In other words, the Monotonicity Lemma implies that splitting and restructuring never decreases the precision of the data-flow solution. It might not necessarily improve the data-flow analysis solution at all nodes, but it never degrades the analysis result at any node. This result differs from the Efficacy Theorem (Theorem 3) since it applies to the data-flow solution of all nodes, and not just those influenced by the particular destructive merge. A very similar result is proved in [2]. In fact, the above proof does not make use of any properties specific to the Split Automaton. Though simple, this lemma will be central in proving the Efficacy result for multiple destructive merges.

Theorem 5. (Efficacy Theorem for Multiple Destructive Merges) *Consider a CFG P with destructive merges $m_1, m_2, \ldots, m_k$ and corresponding split automata $A_1, A_2, \ldots, A_k$. If node $n \in$* `influenced_nodes`(m_x)*, $1 \leq x \leq k$, then in the Split Graph S, node $n_i^x, i \neq 0$ can be optimized, where $S = P \times A_1 \times A_2 \times \ldots \times A_k$.*

Proof. Without loss of generality, let us consider the destructive merge m_1 and the corresponding Split Automaton A_1.

Consider the Split Graph

$$S_1 = P \times A_1 \tag{3.14}$$

If node $n \in$ `influenced_nodes`(m_1), using Efficacy Theorem for single destructive merge (Theorem 3) we can say that some $d_1 \in$ `useful_dff`(n) holds at *in* of node $n_i, i \neq 0$ in Split Graph S_1.

Now consider the Split Graph

$$S = P \times A_1 \times A_2 \times \ldots \times A_k \tag{3.15}$$

Substituting Equation 3.14 in Equation 3.15 we get

$$S = S_1 \times A_2 \times \ldots \times A_k \tag{3.16}$$

Using the Monotonicity Lemma (Lemma 4), we can say that for each node in S say n_q which corresponds to the node $n_i, i \neq 0$ in S_1,

$$\texttt{in\_dff}(n_i) \preceq \texttt{in\_dff}(n_q) \tag{3.17}$$

Thus,

$$d_1 \preceq \texttt{in\_dff}(n_q) \tag{3.18}$$

This implies that each node n_q can be optimized even in the Split Graph S.

□

3.2 Back to Backward Analysis

We now turn our attention to improving the precision of backward data-flow analysis such as liveness analysis. One obvious difference is that for backward analysis the flow of information is not along, but against, the flow of control. Hence, destructive merges are control-flow forks not control flow joins. Thus, to a certain extent the extension from forward to backward analysis is straight-forward: instead of targeting control flow joins where information is lost, we target control flow forks where data-flow information is merged or approximated and data-flow precision is lost. This is reflected in the definition of a destructive merge for backward data-flow analysis (Definition 15).

As before, we use a concrete data-flow analysis to explain the concepts, though our technique is applicable to any backward data-flow analysis. We consider Liveness Analysis.

3.2.1 Computing The Region Of Influence

Single Destructive Merge

In this section, we discuss which nodes can and should be duplicated in order to eliminate a destructive merge for backward analysis. The definitions for Useful Data-flow Facts (Definitions 24 and 25), Influenced Nodes (Definition 26), Revival Data-flow facts (Definition 27) apply both for foward data-flow analysis and backward analysis.

Example. Consider the program in Figure 3.24 and the problem of Liveness Analysis. Control-flow fork D is a destructive merge since variable x is live along outgoing edge $(D - E)$ and is not live along outgoing edge (D, F). This, in turn, makes variable x live at nodes A and C. Thus, the assignment at node these nodes cannot be eliminated.

$$\texttt{useful\_dff}(C) = \{\texttt{x} = \texttt{N}\},$$

$$\texttt{useful\_dff}(D, C) = \{\texttt{x} = \texttt{N}\}.$$

This is due to the fact that if variable x is not live at node D, then x would not be live at node C. This would enable us to optimize node C by removing the statement from node C. Similarly,

$$\texttt{useful\_dff}(A) = \{\texttt{x} = \texttt{N}\},$$

$$\texttt{useful\_dff}(D, A) = \{\texttt{x} = \texttt{N}\}.$$

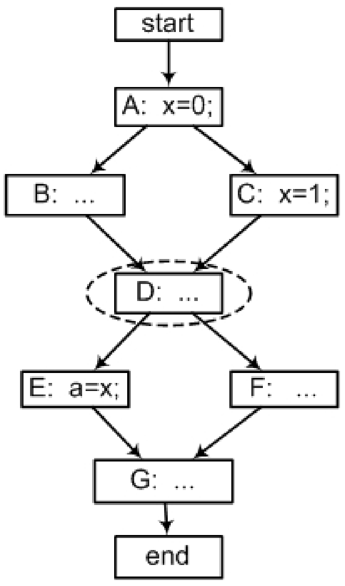


Figure 3.24: Simple acyclic program $P6$. Node D is a destructive merge. The statement at node B cannot be removed since variable x is live.

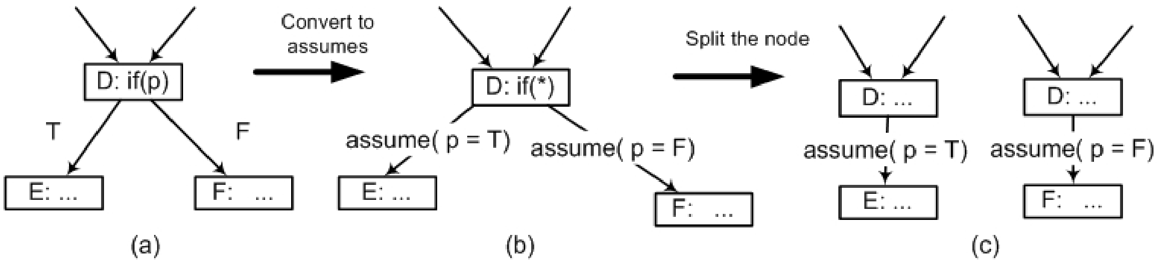


Figure 3.25: Duplication required to eliminate a destructive merge for backward analysis.

Thus,

$$\texttt{influenced\_nodes}(D) = \{A, C\},$$

$$\texttt{destroyed\_dff}(D) = \{\texttt{x} = \texttt{N}\},$$

$$\texttt{revival\_dff}(D) = \{\texttt{x} = \texttt{N}\}.$$

□

Figure 3.25 shows the type of duplication required to eliminate a destructive merge D for backward analysis. We first convert the *if* statements to *assume* statements (Definition 2.1) and then duplicate the control-flow fork which is a destructive merge. Note that during duplication the number of

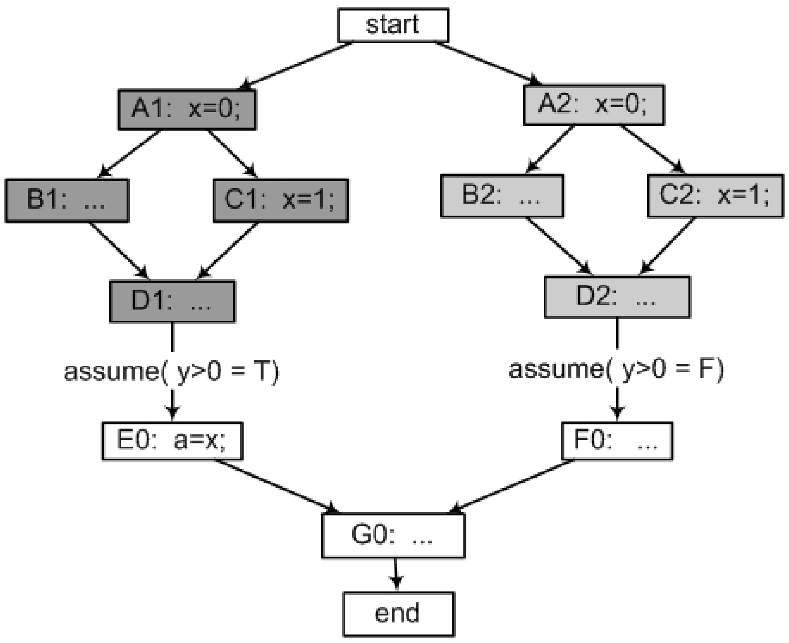


Figure 3.26: The acyclic program of Figure 3.24 after we eliminate the destructive merge at node D by directly applying the techniques developed for forward analysis. The *if* statement has been replaced by the corresponding *assume* statement. The statements at node $C2$ and $A2$ can be removed since variable x is not live. But the program control-flow is non-deterministic at the *start* node.

incoming edges for node D remains the same while the number of outgoing edges decreases.

Though at first glance there doesn't appear to be any difficulty in extending our technique to backward analysis, we show that there are a few subtle issues which need to be handled to ensure correctness of the transformation. To illustrate this difference we look at Figure 3.24. We have already identified that the node D is a destructive merge. Suppose we *reverse* this CFG as shown in Figure 3.27 (a) and replace the *If* node with the appropriate *assume* statements. Then we carry out the same transformation we did earlier for forward analysis. Nodes D, B, C, A are duplicated and the destructive merge at node D is eliminated. The resulting Split Graph is shown in Figure 3.27 (b). Reversing the directions of the edges once more we get the CFG of Figure 3.26 in which the destructive merge at node D present in program $P6$ (Figure 3.24) is eliminated. As we can see, the data-flow precision has indeed improved at nodes $A2$ and $C2$ since variable x is not live at nodes $A2$ and $C2$. We can now eliminate the assignment statements at nodes $A2$ and

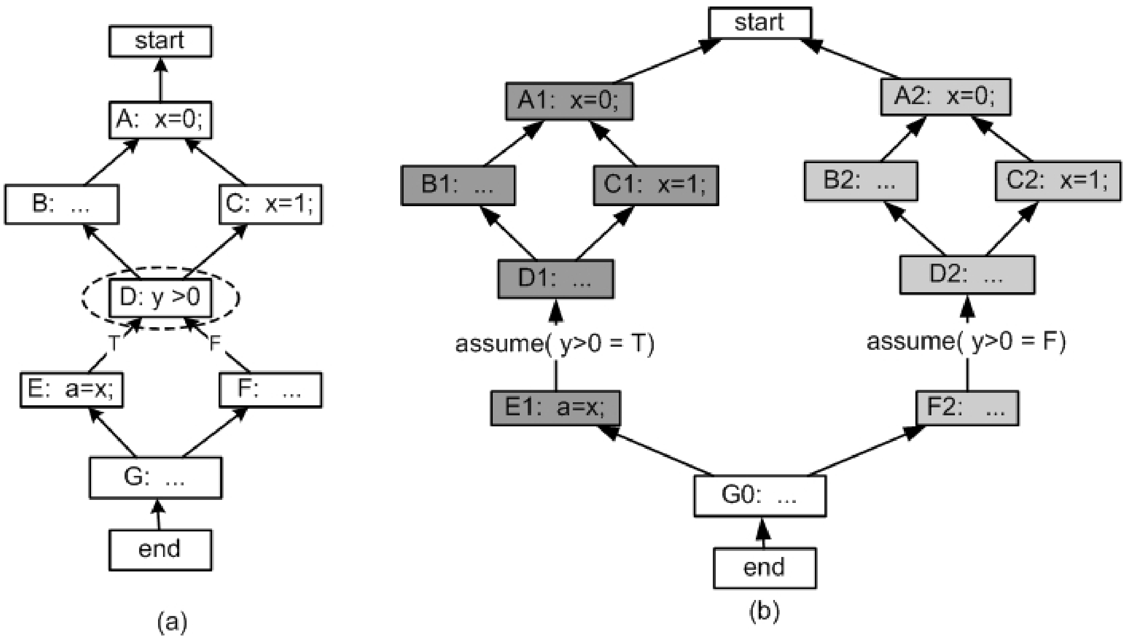


Figure 3.27: *(a)* The reverse graph of the program $P6$ of Figure 3.24. Node D is a destructive merge, *(b)* The split graph after the destructive merge in *(a)* has been eliminated by using the transformation described for forward analysis.

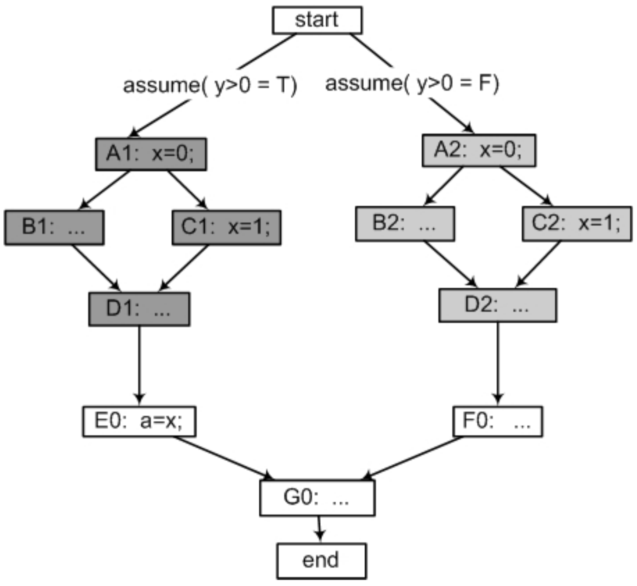


Figure 3.28: Graph obtained after hoisting the *assume* statements in Figure 3.26 to the outgoing edges of the *start* node.

$C2$.

But looking at the control-flow graph we realise that the flow of control has now become *non-deterministic*! This is because the *assume* statements are still placed on edges $(D1 - E1)$ and $(D2 - E2)$. At the start node, it is uncertain as to whether the flow of control should go down the edge $(start, A1)$ or the edge $(start, A2)$.

The problem arises because of the fact that there is some semantic meaning associated with edge (D, E) being taken viz. the predicate $(y > 0)$ evaluates to *True*. Similarly, edge (D, F) being taken implies that $(y > 0)$ evaluates to False. Thus, when we split node D we have to preserve this semantic meaning. In order to ensure that the program remains deterministic, the transformation should hoist the evaluation of the predicate from node D and place it at the appropriate place. For our example, we need to hoist the the evaluation of the predicate to before nodes $A1$ and $A2$. The way we do this is to lift the *assume* statements to edges $(start, A1)$ and $(start, A2)$ as shown in Figure 3.28. We then convert the *assume* statments into the corresponding *if* statement as shown in Figure 3.29.

But the evaluation of the predicate cannot be hoisted to any node. Care should be taken so that the evaluation of the predicate at the new node is the same as that at the original node. For instance, it is not possible to lift the predicate over a statement which modifies the predicate expression.

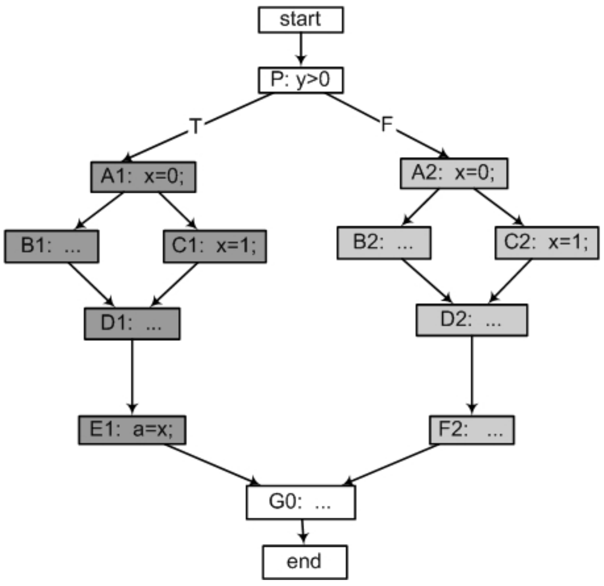


Figure 3.29: Graph obtained after converting the *assume* statement present in Figure 3.28 into the appropriate *if* statement. The flow of control is now deterministic.

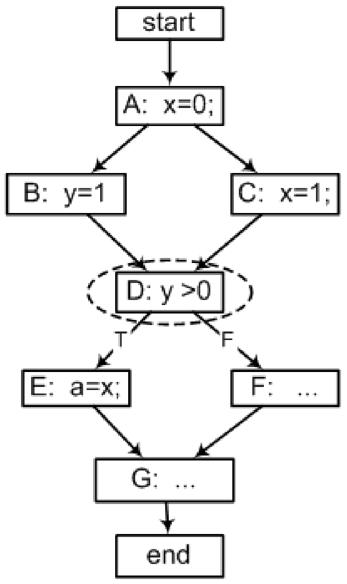


Figure 3.30: Program $P7$.

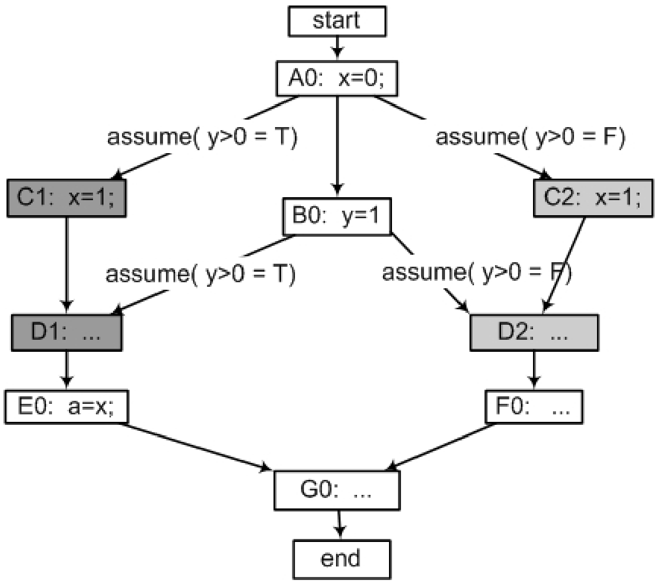


Figure 3.31: Split Graph $S7$ with assume statements.

Consider the program $P7$ in Figure 3.30. The predicate $(y > 0)$ cannot be lifted over block B because the statement $y = 1$ modifies the predicate expression. Similarly, predicate $(y > 0)$ cannot be lifted above block A. Thus, even though it would be beneficial to hoist the predicate evaluation above node A, the predicate evaluation can only be hoisted up to edges (B, D) and (A, C).

Keeping this restriction in mind, Figure 3.31 shows the transformed control-flow graph with the *assume* statements placed on edges (B, D) and (A, C). Figure 3.32 shows the same program where the *assume* statements have been replaced with the corresponding *if* statements. The precision is improved at node C and the assignment statement can be removed since variable x is not live. Notice that the assignment statement at node A cannot be removed in this program as was done in program $P6$. Thus, even though node A is influenced by the destructive merge at node D, node A cannot be split because we can't hoist the evaluation of the predicate above node A. In general, due to the added constraint of hoisting the check present at the destructive merge, all influenced nodes for a destructive merge cannot be optimized. We call the Influenced Nodes which can be optimized taking into considering this added constraint as *Realisable Influenced Nodes*. We formalise this concept next.

Given a destructive merge, we define the *Hoist Region* as that region within which the evaluation of the predicate can be hoisted. We make use *anticipability analysis* (Section 2.2.3) to define this region.

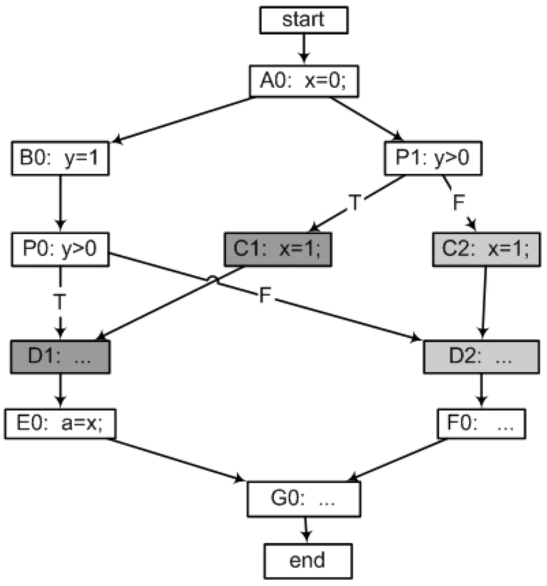


Figure 3.32: Split Graph $S7$ with assume statements replaced with *if* statements.

Definition 36. (Hoist Region) *Given a destructive merge m with predicate p, a node n is in the* Hoist Region *of node m, i.e. $m \in \texttt{hoist\_region}(m)$, iff expression p is anticipatible at the* in *of node n, i.e. $\texttt{ant\_in}(n, p) = 1$.*

Example. In Figure 3.24, node D is the destructive merge with $(y > 0)$ being the predicate.

$$\texttt{ant\_in}(D, (y > 0)) = 1,$$
$$\texttt{ant\_in}(C, (y > 0)) = 1,$$
$$\texttt{ant\_in}(B, (y > 0)) = 1,$$
$$\texttt{ant\_in}(A, (y > 0)) = 1.$$

Thus, $\texttt{hoist\_region}(D) = \{A, B, C, D\}$.

In Figure 3.30, node D is the destructive merge with $(y > 0)$ being the predicate.

$$\texttt{ant\_in}(D, (y > 0)) = 1,$$
$$\texttt{ant\_in}(C, (y > 0)) = 1,$$
$$\texttt{ant\_in}(B, (y > 0)) = 0,$$
$$\texttt{ant\_in}(A, (y > 0)) = 0.$$

Thus, $\texttt{hoist\_region}(D) = \{C, D\}$.

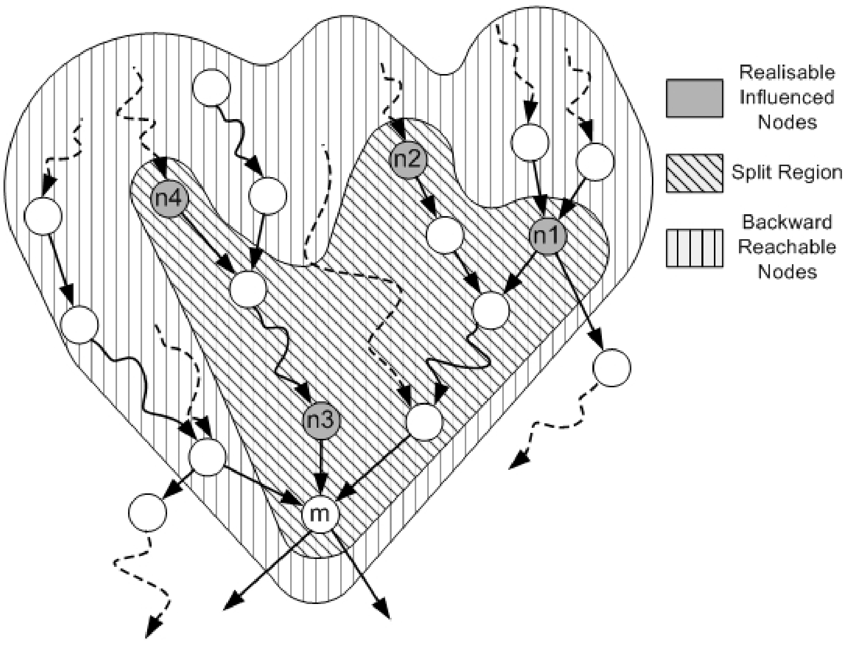


Figure 3.33: Schematic diagram showing the Split Region for a destructive merge m.

□

Taking the Hoist Region into consideration, we define the *Realisable Influenced Nodes* as those Influenced Nodes which are present in the Hoist Region. Accordingly, we define the *Split Region* for backward analysis.

Definition 37. (Realisable Influenced Nodes) *Given a destructive merge m, a node n is a* Realisable Influenced Node , *i.e.* $n \in$ `relisable_influenced_nodes`(m), *iff* $n \in$ `influenced_nodes`$(m) \cap$ `hoist_region`(m).

Similar to the definition of Split Region fo forward analysis, we define the Split Region for backward analysis.

Definition 38. (Split Region) *Given a destructive merge m, a node n is said be in the* Split Region, *i.e.,* $n \in$ `split_region`(m), *iff* $n \in$ `backward_reachable_nodes`(m) *and there exists a node* $u \in$ `realisable_influence_nodes`(m) *and* $u \in$ `backward_reachable_nodes`(n).

Figure 3.33 shows a schematic diagram showing `split_region`(m) for the destructive merge m.

Note that for forward data-flow analysis the Split Region equals the Region of Influence since all influenced nodes are realisable. Due to the inherent nature of the transformation required for backward analysis we have to add these extra constraints to ensure correctness.

Example. Consider Figure 3.24,

$$\begin{aligned}\texttt{influenced\_nodes}(D) &= \{A, C\},\\ \texttt{hoist\_region}(D) &= \{A, B, C, D\},\\ \texttt{realisable\_influenced\_nodes}(D) &= \{A, C\},\\ \texttt{split\_region}(D) &= \{A, B, C, D\}.\end{aligned}$$

Consider Figure 3.30,

$$\begin{aligned}\texttt{influenced\_nodes}(D) &= \{A, C\},\\ \texttt{hoist\_region}(D) &= \{C, D\},\\ \texttt{realisable\_influenced\_nodes}(D) &= \{C\},\\ \texttt{split\_region}(D) &= \{C, D\}.\end{aligned}$$

□

Multiple Destructive Merges

In this section, we extend the concepts of the previous section to handle multiple destructive merges.

Definition 39. (Realisable Influenced Nodes) *Given a set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$, the set of* Realisable Influenced Nodes *is defined as*

$$\texttt{realisable\_influenced\_nodes}(\mathcal{M}) = \bigcup_{m \in \mathcal{M}} \texttt{realisable\_influenced\_nodes}(m)$$

Definition 40. (Split Region) *Given a set of destructive merges $\mathcal{M} = \{m_1, m_2, \ldots, m_k\}$, the* Split Region *corresponding to $\mathcal{M}$ is defined as*

$$\texttt{split\_region}(\mathcal{M}) = \bigcup_{m \in \mathcal{M}} \texttt{split\_region}(m)$$

Extending the definitions to handle multiple destructive merges is straight forward. A node n belongs to $\texttt{realisable\_influenced\_nodes}(\mathcal{M})$ iff it is *influenced* by at least one destructive merge $m \in \mathcal{M}$. Similarly, a node n belongs to $\texttt{split\_region}(\mathcal{M})$ iff it belongs to the *Split Region* of at least one destructive merge $m \in \mathcal{M}$.

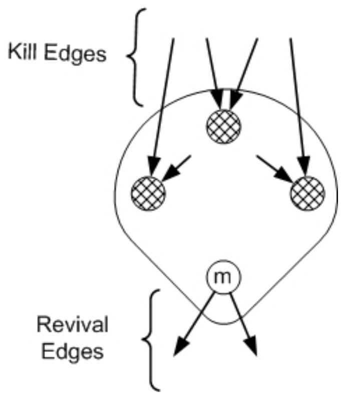


Figure 3.34: Revival and Kill Edges for a Split Region corresponding to a destructive merge m.

3.2.2 The CFG Restructuring

Single Destructive Merge

As before, we use Automata Theory to carry out the transformation and show that the restructured CFG satisfies the correctness and efficacy criteria. Again, we define a Split Automaton which we use to perform the restructuring.

Definition 41. (Kill Edges) *Given a Split Region for a destructive merge m, an edge $e = (u, v)$ is a* Kill Edge, *i.e. $e \in \texttt{kill\_edges}(m)$, iff $u \notin \texttt{split\_region}(m)$ and $v \in \texttt{split\_region}(m)$.*

Thus, Kill Edges are those edges whose target node is in the Split Region and source node is not in the Split Region for a destructive merge m.

Definition 42. (Revival Edges) *For a destructive merge m, all outgoing edges are said be* Revival Edges.

Notice that the definition of Revival Edges for backward analysis differs significantly from that for forward analysis. This new definition is motivated by the need to add the appropriate *assume* statements. Further, let $d_1, d_2, \ldots d_k$ be the k *distinct* data-flow facts for destructive merge m. We can partition the outgoing edges of destructive merge m into k equivalence classes $R_1, \ldots, R_k$. An edge $(m, u) \in R_i, 1 \leq i \leq k$ if $\texttt{in\_dff}(u) = d_i$.

Figure 3.34 shows a schematic diagram illustrating the above concepts.

Example. In Figure 3.24,

$$\texttt{in\_dff}(E) = \{\texttt{x} = \texttt{L}\},\ \texttt{in\_dff}(F) = \{\texttt{x} = \texttt{N}\},$$

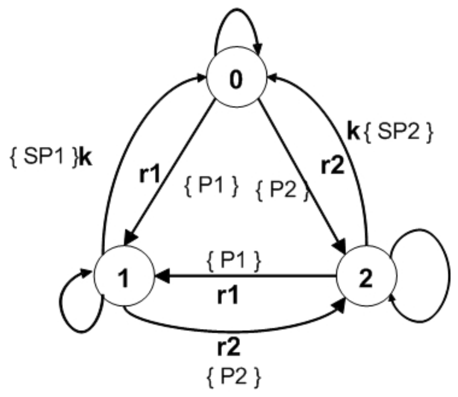


Figure 3.35: The split automaton A_m for destructive merge m. State 0 is the start and accepting state. $r1$ and $r2$ are Revival edges, and k represents the Kill edges. $P1$ and $P2$ are the *state predicates* for states 1 and 2 respectively. They are determined by the *edge predicates* of the Revival edges $r1$ and $r2$ respectively.

$$\begin{aligned}
\texttt{revival\_dff}(m) &= \{\texttt{x} = \texttt{L}, \texttt{x} = \texttt{N}\},\\
\texttt{revival\_edges}(m) &= \{(D, E), (D, F)\},\\
\texttt{split\_region}(m) &= \{A, B, C, D\},\\
\texttt{kill\_edges}(m) &= \{(\textit{start}, A)\}.
\end{aligned}$$

In Figure 3.30,

$$\texttt{in\_dff}(E) = \{\texttt{x} = \texttt{L}\},\ \texttt{in\_dff}(F) = \{\texttt{x} = \texttt{N}\},$$

$$\begin{aligned}
\texttt{revival\_dff}(m) &= \{\texttt{x} = \texttt{L}, \texttt{x} = \texttt{N}\},\\
\texttt{revival\_edges}(m) &= \{(D, E), (D, F)\},\\
\texttt{split\_region}(m) &= \{C, D\},\\
\texttt{kill\_edges}(m) &= \{(B, D), (A, C)\}.
\end{aligned}$$

□

Figure 3.35 shows the schematic diagram of the Split Automaton for backward analysis. The structure of the Split Automata for forward and backward analysis is the same. The main difference is that each transition in the Split Automaton for backward analysis is additionally annotated with a predicate. In particular, all transitions to state $i, i \neq 0$ on edge e are annotated with the predicate $\texttt{edge\_predicate}(e)$. Furthermore, let $e_1, e_2, \ldots, e_k$ are all the edges which make the automaton transition to state $i, i \neq 0$ and let $p_1, p_2, \ldots, p_n$ be the corresponding edge predicates respectively. The *state predicate* for state i is said be $p_1 \vee p_2 \vee \ldots \vee p_n$.

Definition 43. (State Predicate) *Let $e_1, e_2, \ldots, e_n$ be the edges which cause the split automaton to transition from state 0 to state i and $p_1, p_2, \ldots, p_n$ be the corresponding edge predicates respectively. Then the* state predicate *for state i, i.e.* $\texttt{state\_predicate}(i)$*, is the predicate $p_1 \vee p_2 \vee \ldots \vee p_n$.*

All the self-loops in the split automaton are annotated with the *true* predicate. This is mainly done for uniformity of exposition and for all practical purposes can be treated no-operations. These predicates on the Split Automaton transitions are used in *assume* statements which will be inserted on the edges of the resulting product automaton. The *assume* statements will then be converted into corresponding conditional checks. This two step approach is needed when we deal with multiple destructive merges being eliminated at the same time.

Definition 44. (Split Automaton for Backward Analysis) *The* Split Automaton A_m *corresponding to a destructive merge m is a finite-state automaton defined as:*

- *the input alphabet $\Sigma = E$, the set of all edges in the CFG,*
- *a set of $k+1$ states $Q = \{s_0, s_1, s_2, \ldots, s_k\}$,*
- *$s_0 \in Q$ is the initial and accepting state,*
- *the transition function $\delta : Q \times \Sigma \rightarrow Q$ defined as*
  $(s_i, e) \rightarrow s_j, e \in R_j$ (Revival Transitions)
  $(s_i, e) \xrightarrow{p} s_0, e \in \texttt{kill\_edges}(m), p = \texttt{state\_predicate}(s_i)$ (Kill Transitions)
  $(s_i, e) \rightarrow s_i, otherwise$ (No Transition)
- *the predicate annotation* $\texttt{predicate} : Q \times \Sigma \rightarrow$ *predicate defined as*
  if $(s_i, e) \rightarrow s_j, e \in R_j$ *then* $\texttt{predicate}(s_i, e) = \texttt{edge\_predicate}(e)$
  if $(s_i, e) \rightarrow s_0, e \in R_j$ *then* $\texttt{predicate}(s_i, e) = \texttt{state\_predicate}(s_i)$
  $\texttt{predicate}(s_i, e) = T$ *,otherwise*

Example. Figure 3.36 shows the Split Automaton corresponding to the destructive merge D in Figure 3.30.

$$\texttt{state\_predicate}(1) = y > 0, \quad \texttt{state\_predicate}(2) = !(y > 0).$$

The trasitions to state 1 are labelled with $\texttt{edge\_predicate}((D, E)) = y > 0$,

□

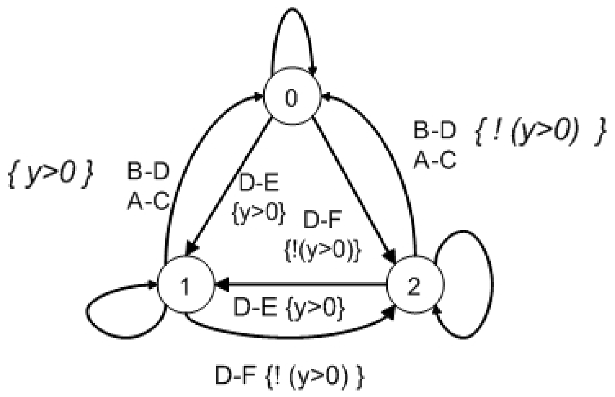


Figure 3.36: The Split Automaton A_D corresponding to destructive merge D in Figure 3.30.

Similar to the case for forward analysis, we define the Split Graph for backward analysis. The major difference is that we work on the *reverse graph* (Definition 6) of the CFG for Backward analysis.

Definition 45. (Split Graph for Backward Analysis) *Given a CFG P and and a Split Automaton A_m, we define the* Split Graph *S to be the* reverse graph *of $P^r \times A_m$, where P^r is the reverse graph of P and $\times$ is the product automaton operator [13].*

Additionally whenever the Split Automaton transitions from state i to state j on edge a, add an assume(p) *on the corresponding edge in the Split Graph, where $p = \texttt{predicate}(i, a)$.*

Example. Figure 3.31 shows the Split Graph $S7$ constructed when the program $P7$ in Figure 3.30 is split using the Split Automaton of Figure 3.36. Notice the *assume* statement placed on edge $(D1, B0)$. The predicate for the assume is the state predicate for state 1 in the Split Automaton.

□

Figure 3.2.2 shows the algorithm to construct the Split Graph for Backward analysis for a single destructive merge. The main differences between the forward and backward analysis is that we first have to reverse the input CFG (Line 1 before performing the product. Further, during the product computation we have to add the appropriate assume statement on the edge. This is shown on Line 18. Finally, after the product is computed, the algorithm constructs the reverse graph of the ouput Product graph as seen in Line 22.

```
BkwdSplit( $G$,  $A_1(Q_1, \Sigma, \delta_1, s_1, F_1)$)
      // Construct the reverse graph.
 1    $G^r \leftarrow reverse(G)$
      // The start node of product automaton composed of
      // start nodes of split automaton and reverse graph .
 2    $s_P \leftarrow (end, s_1)$
      // The states of the product automaton; initially only contains the start state.
 3    $Q_P \leftarrow s_P$
      // The final nodes of the product automaton.
 4    $F_P \leftarrow \{start\} \times F_1$
      // Compute the transition function for the product automaton.
      // The transition function of the product automaton; initially empty.
 5    $\delta_P \leftarrow \emptyset$
      // Worklist of states; initially contains the start state.
 6    WorkList $W \leftarrow s_P$
      // Do while the worklist is not empty.
 7    While $W \neq$ EMPTY
 8        do
 9            $state \leftarrow$ get($W$)  // Pop the next state from the worklist.
10            $(n_1, q_1) \leftarrow state$
              // Foreach letter in the input alphabet,compute the next state.
11              Foreach $a \in \Sigma$
12                do  // Get the next state.
13                  $q_1` \leftarrow \delta_1(q_1, a)$
14                  $next\_state \leftarrow (\delta_P(n_1, a), q_1`)$
15                  If ($next\_state \notin Q_P$)  // Check if this is a new state.
16                    then
17                        // Add the assume statement to edge.
18                        $edge(state, new\_state) \leftarrow assume(\texttt{predicate}(q_1, a))$
                          // Add the new state to the set of states.
19                        $Q_P \leftarrow Q_P \cup \{ next\_state \}$
                          // Push the new state onto the worklist.
20                        put($W, next\_state$)
                  // Add the new transition to the transition function.
21                  $\delta_P \leftarrow \delta_P \cup \{ (state, next\_state) \}$
22            $P^r \leftarrow reverse(P)$  //Reverse the product graph
23            return $P^r$
```

Figure 3.37: Algorithm to compute the Split Graph for Backward Analysis for a single destructive merge.

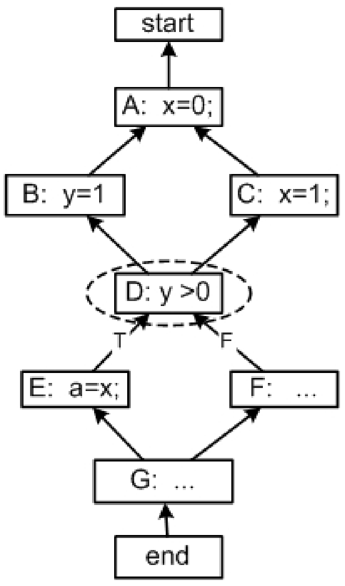


Figure 3.38: Reverse graph of the Program $P7$ in Figure 3.30.

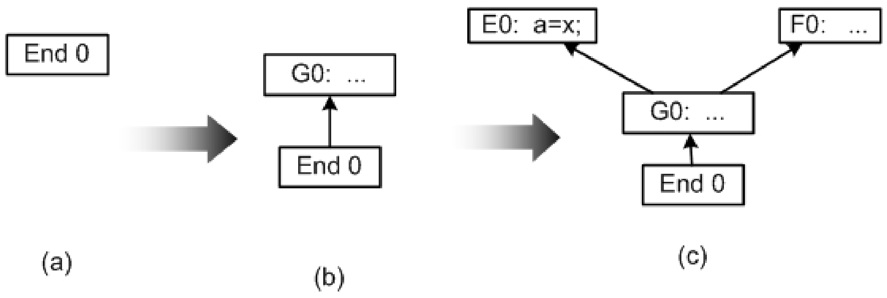
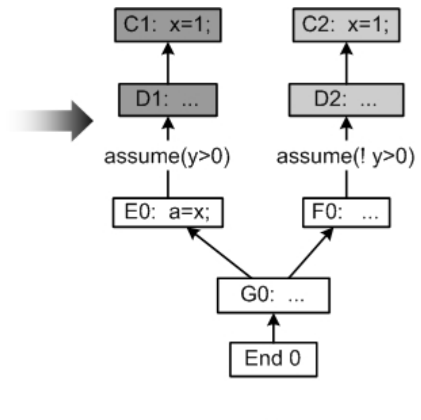


Figure 3.39: An example of Product computation for backward analysis.

Example. We now illustrate the workings of the the above algorithm using the program $P7$ of Figure 3.30. Figure 3.38 shows the reverse graph of the program $P7$ as computed at Line 1. We now compute the product of this reverse graph and the Split Automaton of Figure 3.36.

Figure 3.39 shows the *start* state of the product automaton. It is composed of the *start* state of the *reverse* graph and the start state of the Split Automaton. Notice that the *start* state of the *reverse* graph is the *end* node of the original CFG.

The Split Automaton remains in state 0 on following the edge (G, end). Thus, as seen in Figure 3.39(b), a new state $end\,0$ is created and the transition $(end\,0, G0)$ is added to the product automaton. In a similar manner, states $E0$ and $F0$ are created as seen in Figure 3.39(c). For clarity, we are not



(d)

Figure 3.40: An example of Product computation for backward analysis. *(contd.)*

showing the trivial *assume(T)* statements on the edges.

Consider the state $E0$ in the product automaton. The Split Automaton is in state 0 and the CFG is in node E. On seeing the edge (D, E), the split automaton transitions to state 1 and the CFG goes to state D. Further, the predicate $y > 0$ is used to create the $assume(y > 0)$ statement on edge $(D1, E0)$ in the product automaton. Further, state $D1$ transitions to state $C1$. A similar situation occurs for the state $F0$. as illustrated in Figure 3.40(d).

In state 1 when the automaton sees the edge (B, D) it transitions to state 0 and the corresponding predicate is $y > 0$. Thus, we see the transition from state $D1$ to state $B0$ in the product automaton with the *assume* statement $assume(y > 0)$ add on the edge. The same logic applies to state $D2$ of the automaton when it see the edge (B, D). This is seen in Figure 3.41(e).

Finally, Figure 3.42(f) shows the completed product automaton. Notice that in this CFG each edge is in the reverse direction.

Figure 3.43(g) shows the *reverse* graph of the graph in Figure 3.42(f). This is the final Split Graph returned by the algorithm in Figure 3.2.2.

□

The remaining step in the approach for backward analysis is that of converting the *assume* statements back to the corresponding *if* statements. The algorithm is described in Figure 3.2.2. The algorithm takes as input a node n whose outgoing edges have *assume* statements. The goal is to replace these *assume* with the corresponding *if* statements. The trivial case is when the

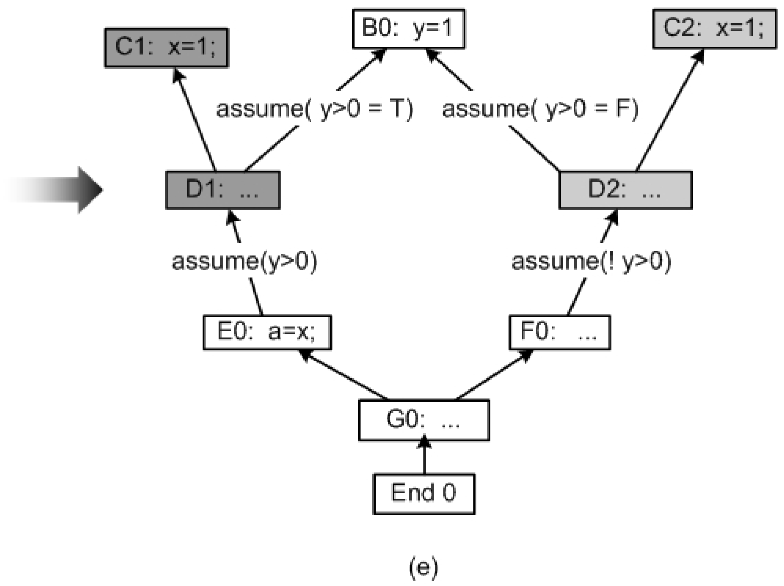


Figure 3.41: An example of Product computation for backward analysis.*(contd.)*

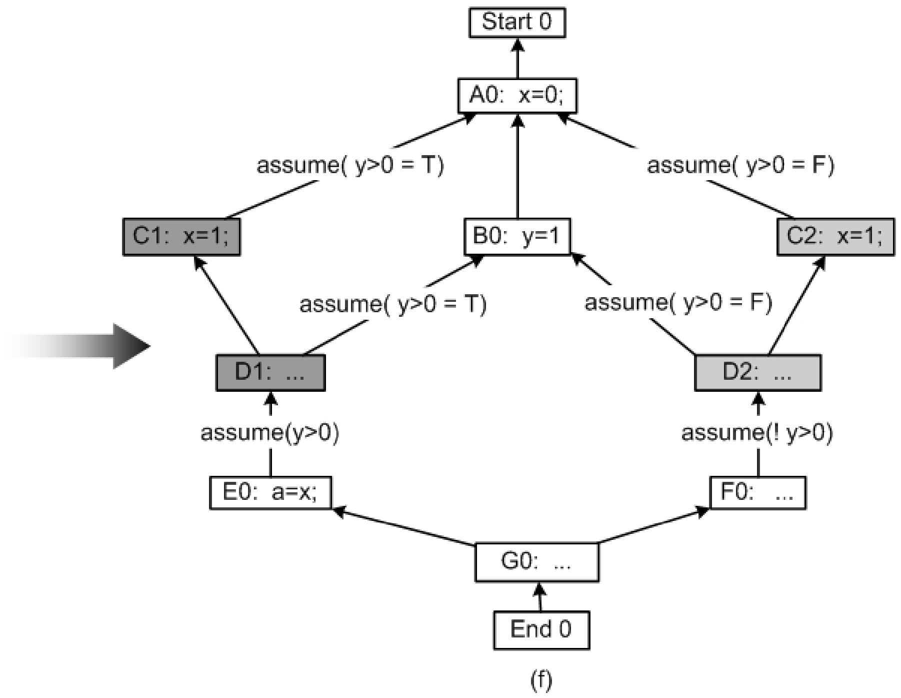


Figure 3.42: An example of Product computation for backward analysis.*(contd.)*

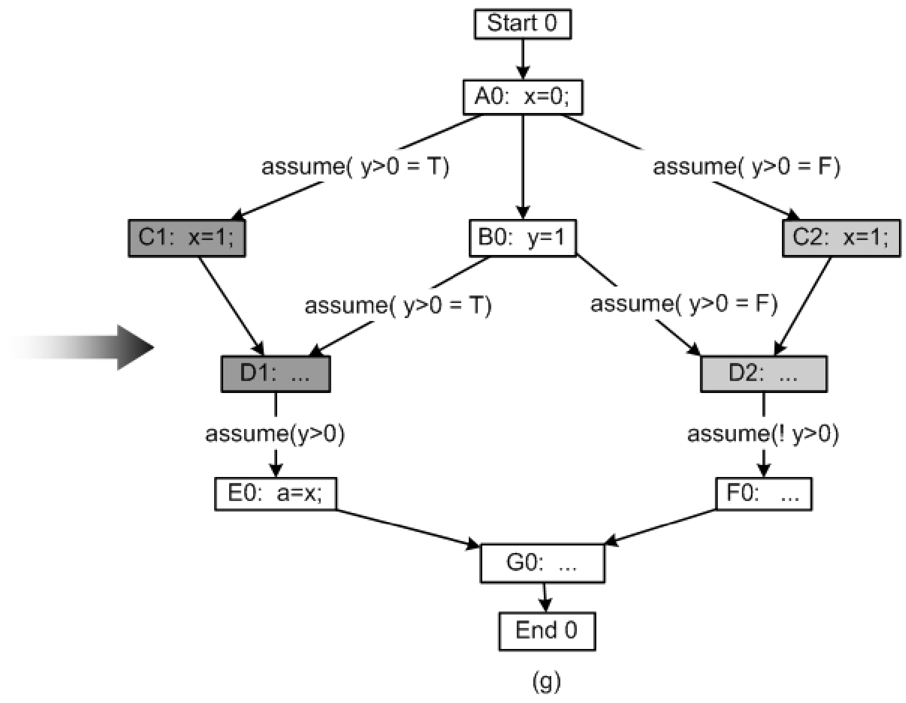


Figure 3.43: The *reverse* graph of the CFG in Figure 3.42.

there is a single outgoing edge for node n. In this case, we can simply remove the *assume* statement as shown in Line 3.

The main loop of the algorithm iterates over all the predicates present in the *assume* statements until all *assume* statements have been replaced. At Line 8, edges (n, m_1) and (n, m_2) are are labelled with assume statements which are same except that (n, m_1) has an $assume(p)$ while (n, m_2) has an $assume(!p)$ i.e. the *assume* statements differ only in the predicate p. This check is carried out at Line 7. Though this might seem an expensive check at first glance, note that since we are at max splitting k destructive merges, there can only be k predicates. Furthermore, the *assume* statements can be representing in the classic bit-vector notation i.e. if an edge has an $assume(p_i)$ then bit i in the vector is set, else it is reset. Using this notation, the check the Line 7 translates to finding two bit-vectors which differ in exactly one place by simply *xor*-ing the bit-vectors.

Having found two such edges, a new *If* node is created and placed into the CFG as seen in Line 9. Care is taken to connect the *true (T)* and *false (F)* edges of the new *If* node depending on the *assume* statements.

Finally, at Line 10, the remaining *assume* statements are placed on the edge between the original node n and the new *If* node n_1. The algorithm

```
ConvertAssumes($n$)
        // Node $n$ has outgoing edges with assume statements.
 1  While assume statements exist
        do // If $n$ has only one outgoing edge
 2          If $|\texttt{succ}(n)| = 1$
               then // Simply remove the assume statement.
 3                  Edge $(n, \texttt{succ}(n)) \leftarrow \emptyset$
 4                  Return
 5          Foreach $p \in \mathit{Predicates}$
               do //For all outgoing edges of the given node $n$.
 6                  Forall edges ( $(n, m_1)$, $(n, m_2)$ )
                       do // If the assume statements differ only in predicate $p$.
 7                        If $(n, m_1) = \mathit{assume}(p) \cdot \mathit{assume}(q)$
                             and $(n, m_2) = \mathit{assume}(!p) \cdot \mathit{assume}(q)$
                             then
 8                               Make new node $n_1$ = If$(p)$.
 9                               Create new edges $(n \rightarrow n_1), (n_1 \xrightarrow{T} m_1), (n_1 \xrightarrow{F} m_2)$.
                                 //Add the remaining assume statements to the new edge
10                               Edge $(n \rightarrow n_1) \leftarrow \mathit{assume}(q)$.
```

Figure 3.44: Algorithm to convert *Assume* statements to *If* statements for node n.

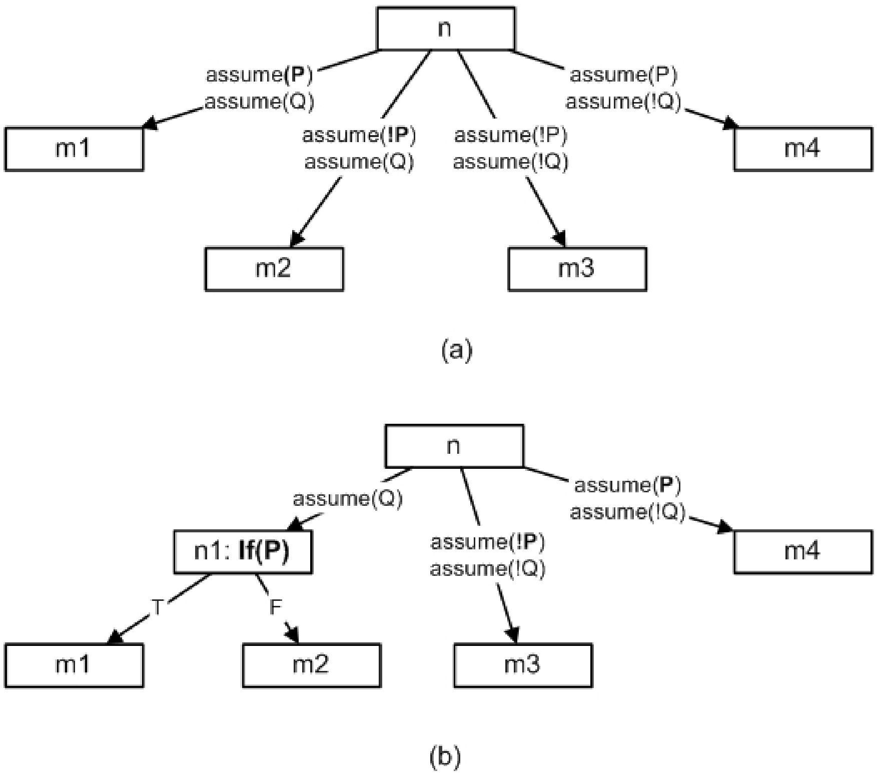


Figure 3.45:

continues until no more *assume* statements remain. The algorihm is sure to terminate since at each stage one predicate p is eliminated.

Example. Figure 3.45(a) shows the original node n which has *assume* statements on it's outgoing edges. We see that edges $(n, m1)$ and $(n, m2)$ satisfy the check at Line 7 of the algorithm. Figure 3.45(a) shows the resulting program after the *assume*(p) is replaced with the *If(p)* nodes $n1$. Since edge $(n, m1)$ had the *assume*(p), edge $(n1, m1)$ is the *true* branch for the *If*. On the other hand, since edge $(n, m2)$ had the *assume*$(!p)$, edge $(n1, m2)$ is the *false* branch for the *If*.

In a similar manner, edges $(n, m3)$ and $(n, m4)$ in Figure 3.45(b) are replaced with the *If* $n2$ in Figure 3.46(c). Finally, edges $(n, n1)$ and $(n, n2)$ are chosen and the *assume(Q)* is replaced by the *If(Q)* node $n3$. In this way, all the *assume* statements are replaced by the corresponding *If* statements as shown in Figure 3.46(d).

□

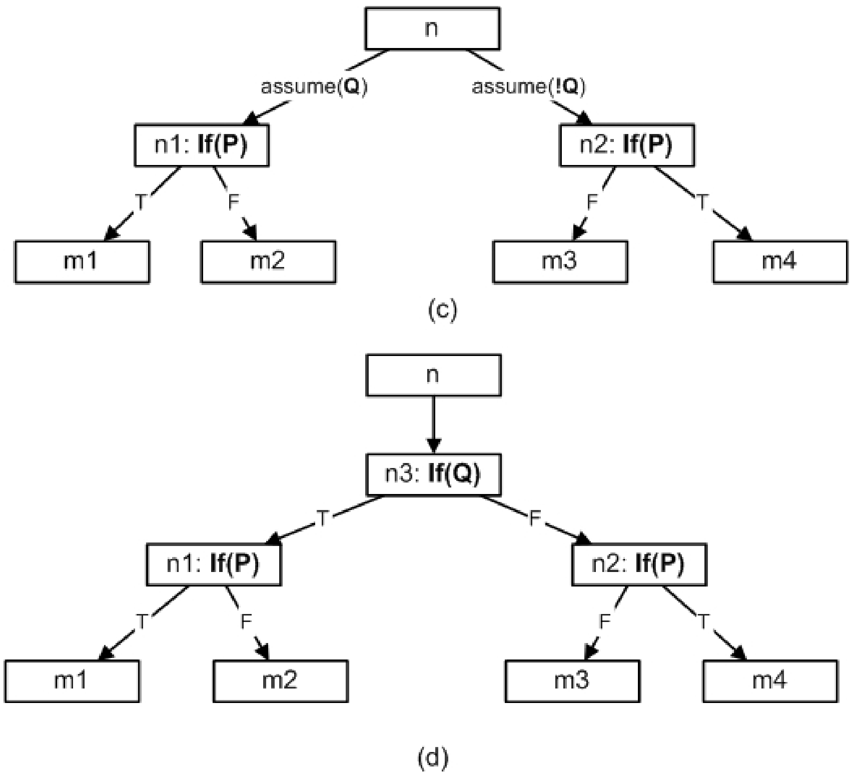

n
assume(Q)
assume(!Q)
n1: If(P)
n2: If(P)
T
F
F
T
m1
m2
m3
m4
(c)
n
n3: If(Q)
T
F
n1: If(P)
n2: If(P)
T
F
F
T
m1
m2
m3
m4
(d)

Figure 3.46:

Example. The algorithm is also used to convert the *assume* statements in Figure 3.31 into *If* nodes as shown in Figure 3.32.

□

Using Slicing

Since the Hoist Region affects the extent to which our technique is applicable, we use a restricted form of *static backward slicing* to not only hoist the evaluation of the predicate p, but also other statements in the backward slice of p. The backward slice is restriced in the sense that we only consider the part of the backward slice which is present in the *same basic block* as the predicate evaluation p. This implies that only straight-line code is hoisted which do not contain complex control-flow. Though theoretically the extension for hoisting more complex backward-slices is possible, the implementation becomes unnecessarily complex. This is similar in flavour to the slicing transformation described in [5]. The main difference is that the method is extremely simple and clean. Instead of just the predicates which are placed on the edges of the Split Automaton, the restricted backward slice is placed. Thus, instead of placing only *assume* statements, we also have to place this restricted backward slice.

Thus, the anticipability analysis which is carried out has to be done on the *upward exposed expressions* [15] in the backward slice of the the predicate p.

Example. Consider the basic block
1. $b = c;$
2. $z = 2;$
3. $y = a + b;$
4. $z = z + 1;$
5. $If(y > 0)$

The predicate is $y > 0$. The backward slice for the expression $y > 0$ contains the statements
1. $b = c;$
3. $y = a + b;$
5. $If(y > 0)$

The expressions which are *upward exposed* are a and c. y and b are not *upward exposed* since there are definitions of y and b in the same basic block.

Note that since y is modified in the same basic block, it is not possible to hoist the evaluation of the predicate $y > 0$ over this basic block. Using the backward slice of the predicate and hoisting all the statements in it, the hoist region increases in size. Thus, the scope for optimization increases.

□

Multiple Destructive Merges

Similar to forward analysis, it is easy to extend the above approach to eliminating multiple destructive merges. Consider the set of destructive merges $\mathcal{M} = \{m_1, m_2, \dots, m_k\}$ which are to be eliminated. Let $A_1, A_2, \dots, A_k$ be the corresponding Split Automata (Definition 44).

Definition 46. (Split Graph for multiple Destructive Merges for Backward Analysis) *Given a CFG P and and a set of Split Automata $A_1, A_2, \dots, A_k$, we define the* Split Graph *S to be* reverse graph *of $P^r \times A_1 \times A_2 \times \dots \times A_k$, where $\times$ is the product automaton operator [13].*

Additionally whenever the Split Automaton transitions from state i to state j on edge a, add an assume(p) *on the corresponding edge in the Split Graph, where $p =$ `predicate`(i, a).*

It is simple to extend the algorithm described in Figure 3.2.2 to handle multiple destructive merges. The efficacy and equivalence results also hold when eliminating multiple destructive merges. The proofs have the same flavour as that for Forward analysis. As for forward analysis, our approach for backward analysis does not restrict the structure of the CFG. The approach seamlessly handles loops as well.

Chapter 4

Tradeoff

Economy is not how little one can spend,
but how wisely one can spend it.
Origin unknown

I think you can achieve anything, if you are
willing to pay the price
VINCE LOMBARDI

The discussion till now has not dealt with the trade-off between the data-flow precision achieved and the increase in the code size due to the restructuring inherent to this approach. We have seen that the benefit of eliminating a destructive merge m is related to the number of nodes *influenced* by node m. Also, the size of the *Region of Influence* of the destructive merge determines the resulting cost in terms of increase code size. Thus, this trade-off depends directly on the set of destructive merges we choose to eliminate to form the Split Graph. Thus, we need to pick the best set of destructive merges which maximizes the benefit in terms of data-flow precision, for a given cost in terms of code size. In this section, we will prove that this problem is NP-Hard. We then proceed to describe certain heuristics for the same.

4.1 Theoretical Analysis

Before we move to the NP-Hardness result, we define the notion of independece among destructive merges.

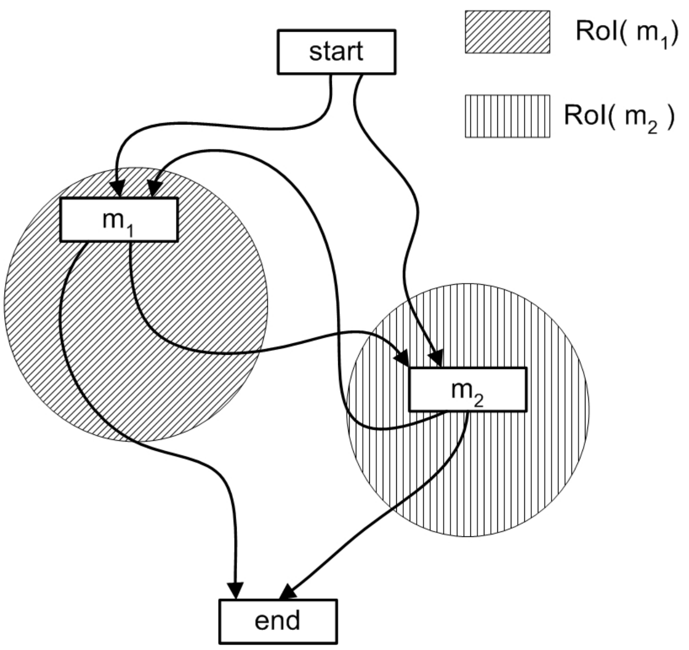


Figure 4.1: Schematic diagram illustrating two independent destructive merges.

Definition 47. (Independent Destructive Merges) *Two destructive merges m_1 and m_2 are said to be* independent *iff*

$$\texttt{RoI}(m_1) \cap \texttt{RoI}(m_2) = \emptyset.$$

Example. Consider Figure 4.1. Destructive merges m_1 and m_2 are independent since their respective Regions of Influence do not share any node in common.

□

Independent destructive merges are independent is the sense that we can analyze the code size increase due to restructuring separately, and combine them using simple addition. This notion is formalised as the Additive Property of independent destructive merges.

Theorem 6. (Additive Property) *Consider a CFG G and two* independent *destructive merges m_1 and m_2 with their respective split automata A_1 and A_2, then*

$$|\ N(G)\ | + |\ N(G \times A_1 \times A_2)\ | = |\ N(G \times A_1)\ | + |\ N(G \times A_2)\ |$$

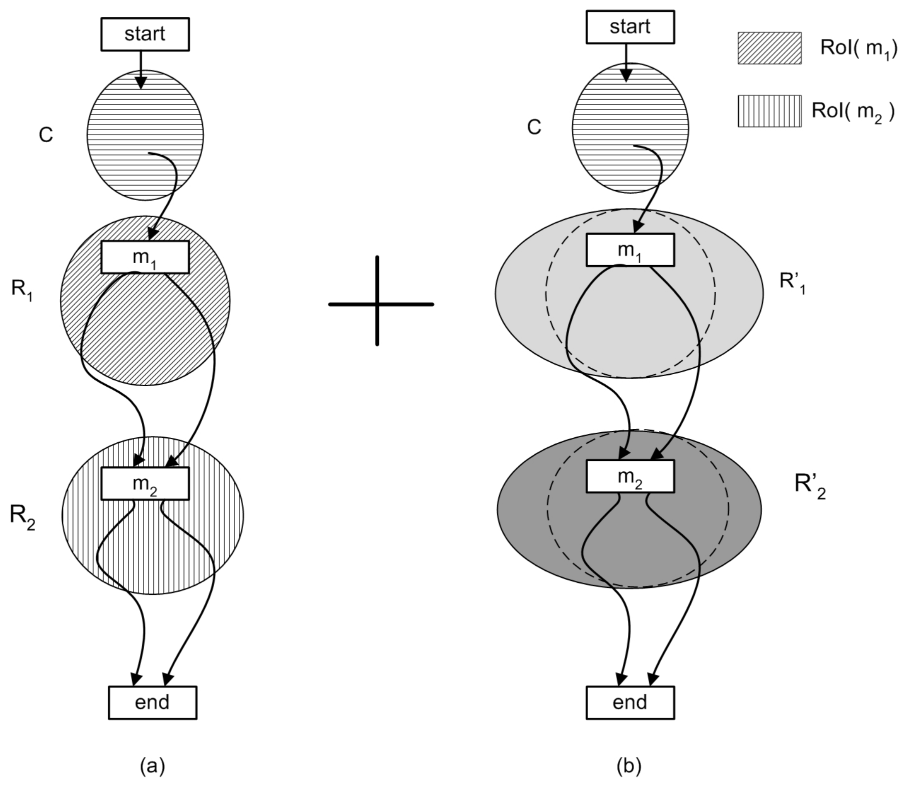


Figure 4.2: Schematic diagram illustrating the LHS of Equation 4.1. *(a)* CFG G, *(b)* Split Graph $S \overset{\texttt{def}}{=} G \times A_1 \times A_2$.

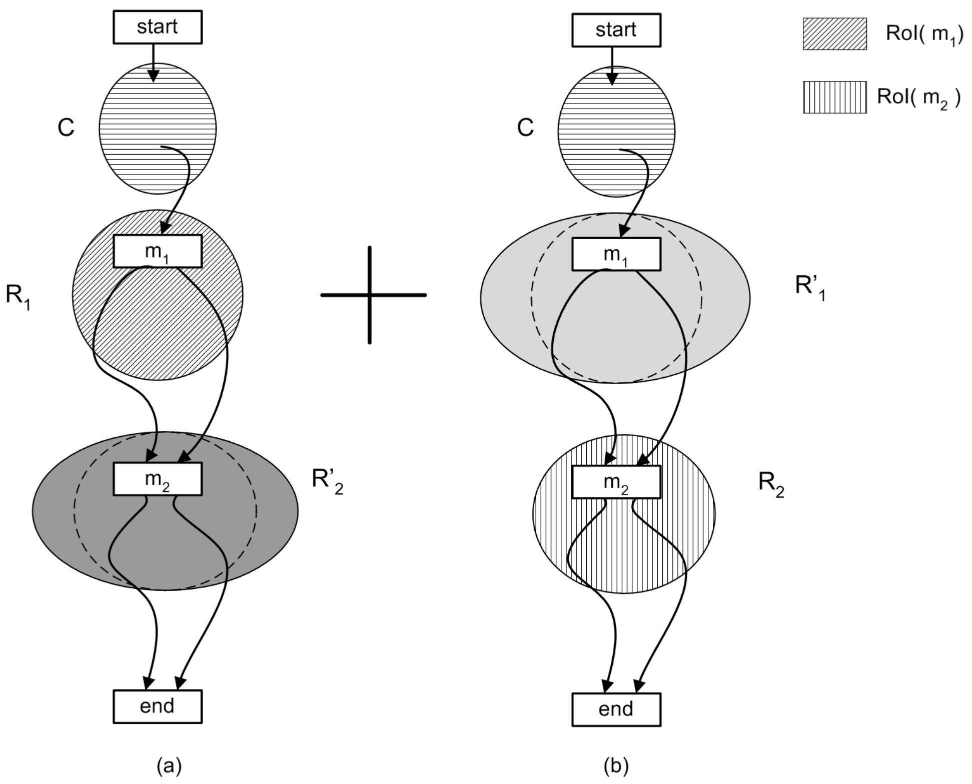


Figure 4.3: Schematic diagram illustrating the RHS of Equation 4.1. *(a)* Split Graph $S_1 \overset{\texttt{def}}{=} G \times A_1$, *(b)* Split Graph $S_2 \overset{\texttt{def}}{=} G \times A_2$.

Proof. Consider the equation,

$$\mid N(G) \mid + \mid N(G \times A_1 \times A_2) \mid = \mid N(G \times A_1) \mid + \mid N(G \times A_2) \mid \tag{4.1}$$

Let

$$S \overset{\texttt{def}}{=} G \times A_1 \times A_2,$$
$$S_1 \overset{\texttt{def}}{=} G \times A_1,$$
$$S_2 \overset{\texttt{def}}{=} G \times A_2,$$
$$R_1 \overset{\texttt{def}}{=} \texttt{RoI}(m_1),$$
$$R_2 \overset{\texttt{def}}{=} \texttt{RoI}(m_2).$$

Since the two destructive merges are independent, in Split Graph S_1 no node in R_2 will be split, while nodes in R_1 will be duplicated to form nodes R'_1. Similarly, in Split Graph S_2 no node in R_1 will be split, while nodes in R_2 will be duplicated to form R'_2. This is illustrated in Figure 4.3. Further, in Split Graph S nodes in R_1 and R_2 will be duplicated due to elimination of destructive merges m_1 and m_2 respectively to form the same nodes R'_1 and R'_2 as show in Figure 4.2. Let C represnt those nodes which lie outside both R_1 and R_2 and hence are unaffected by any restructuring.

$$\mid N(G) \mid = \mid C \mid + \mid R_1 \mid + \mid R_2 \mid \tag{4.2}$$

$$\mid N(G \times A_1 \times A_2) \mid = \mid C \mid + \mid R'_1 \mid + \mid R'_2 \mid \tag{4.3}$$

Using Equations 4.2 and 4.3, the LHS of Equation 4.1 can be written as

$$\mid N(G) \mid + \mid N(G \times A_1 \times A_2) \mid = (\mid C \mid + \mid R_1 \mid + \mid R_2 \mid) + (\mid C \mid + \mid R'_1 \mid + \mid R'_2 \mid) \tag{4.4}$$

Similarly, we have

$$\mid N(G \times A_1) \mid = \mid C \mid + \mid R'_1 \mid + \mid R_2 \mid \tag{4.5}$$

$$\mid N(G \times A_2) \mid = \mid C \mid + \mid R_1 \mid + \mid R'_2 \mid \tag{4.6}$$

Using Equations 4.5 and 4.6, the RHS of Equation 4.1 can be written as

$$\mid N(G \times A_1) \mid + \mid N(G \times A_2) \mid = (\mid C \mid + \mid R'_1 \mid + \mid R_2 \mid) + (\mid C \mid + \mid R_1 \mid + \mid R'_2 \mid) \tag{4.7}$$

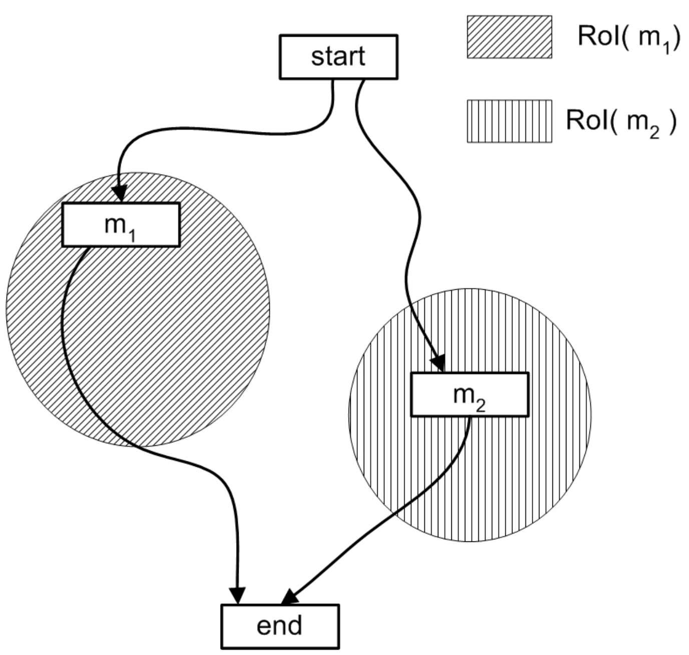


Figure 4.4: Schematic diagram illustrating the sufficiency condition for independent destructive merges.

Rearranging Equation 4.7, we get

$$| N(G \times A_1) | + | N(G \times A_2) | = (| C | + | R_1 | + | R_2 |) + (| C | + | R'_1 | + | R'_2 |) \tag{4.8}$$

From Equations 4.4 and 4.8, we see that

$$| N(G) | + | N(G \times A_1 \times A_2) | = | N(G \times A_1) | + | N(G \times A_2) |$$

This proves the Additive Property for independent destructive merges.

□

What independent destructive merges allow us to do is to calculate the increase in code size caused due to restructuring for each destructive merge separately, and then by simply adding these up we get the total increase in code size due to all destructive merges used together.

We now state a sufficient, but not necessary, condition for two destructive merges to be independent.

Lemma 5. (Sufficient Condition for Independence) *Two destructive merges m_1 and m_2 are independent if*

$$\texttt{reachable\_nodes}(m_1) \cap \texttt{reachable\_nodes}(m_2) = \{\text{end}\}.$$

Proof. From the Region of Influence of a destructive merge (Definition 28) it follows that

$$\texttt{RoI}(m_1) \subseteq \texttt{reachable\_nodes}(m_1) \tag{4.9}$$

$$\texttt{RoI}(m_2) \subseteq \texttt{reachable\_nodes}(m_2) \tag{4.10}$$

We are given that

$$\texttt{reachable\_nodes}(m_1) \cap \texttt{reachable\_nodes}(m_2) = \{end\} \tag{4.11}$$

Note that the *end* node cannot belong to the region of influence since, it has no statements, it cannot be influenced by any destructive merge, and since it has no successors it cannot reach any influenced nodes.

Thus, using Equations 4.9 and 4.10 in Equation 4.11,

$$\texttt{RoI}(m_1) \cap \texttt{RoI}(m_2) = \emptyset$$

This implies that destructive merges m_1 and m_2 are independent. □

Definition 48. *The problem* SPLIT *is defined by the triple* $(P, \mathcal{A}, C)$ *where:*

- P *is a program,*
- $\mathcal{A}$ *is a set of split automata corresponding to the various destructive merges, and*
- C *is maximum increase in code size that is permitted due to restructuring.*

A solution to SPLIT *is a subset* B *of* $\mathcal{A}$ *such that applying* B *to the program* P *does not increase the code-size by more than* C *and which maximizes the number of influenced nodes which can be optimized in the resulting program* P'*.*

Theorem 7. (SPLIT THEOREM) SPLIT *is NP-Hard.*

Proof. We shall reduce KNAPSACK [17] to it. We are given a set $\mathcal{I}$ of n items, each item i having a specific weight w_i and a profit p_i. The goal of KNAPSACK is to pick a subset $J \subseteq \mathcal{I}$ of the items so as to maximize the total profit subject to the condition that the total weight is less than a specified weight W.

Intuitively, in our reduction, picking an item i in KNAPSACK will correspond to selecting an split automaton in the solution of SPLIT . Thus, we construct a program P, in which for each item i in KNAPSACK there exists a destructive merge D_i and a split automaton a_i so that $|\ \texttt{influenced\_nodes}(D_i)\ | =$

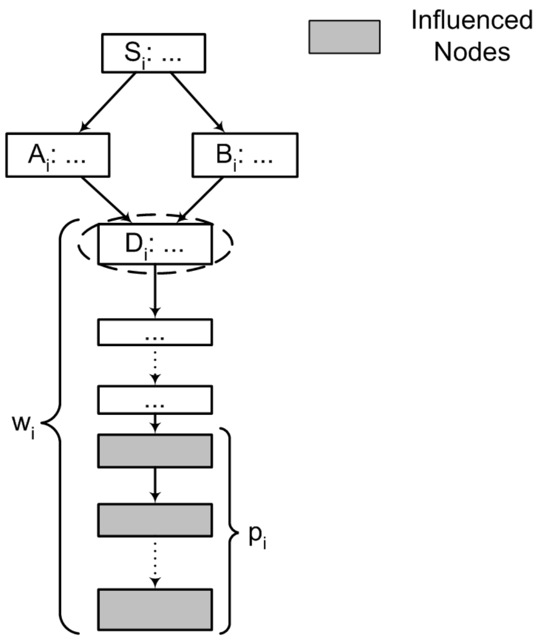


Figure 4.5: The program fragment P_i constructed corresponding the the Knapsack item i. D_i is a destructive merge. The number of influenced nodes is p_i and the size of the region of influence is w_i.

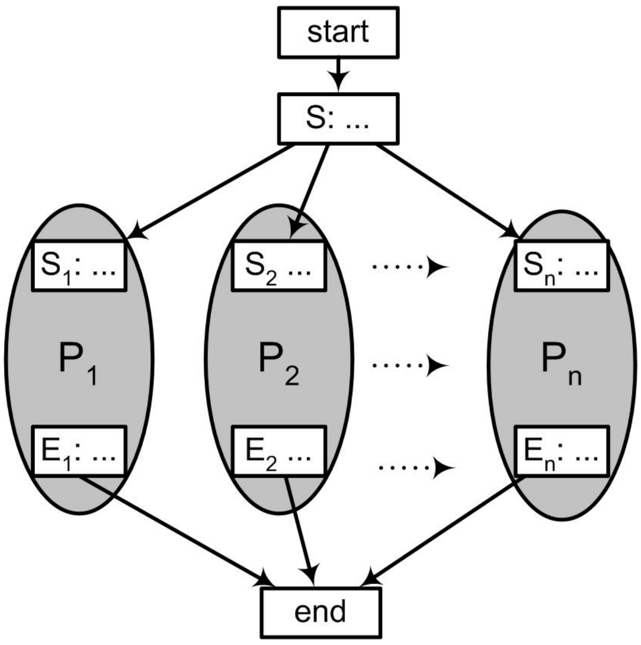


Figure 4.6: The structure of program P constructed from the Knapsack instance. Program P is composed of fragments P_i as shown in Figure 4.5.

p_i and $|\,\texttt{RoI}(D_i)\,| = w_i$. Such a program fragment P_i corresponding to an item i is shown in Figure 4.5.

Further, the profits and costs of items in KNAPSACK are independent of each other i.e. the cost of picking an item i does not depend on whether item j has been placed in the knapsack. To ensure this we have to have to construct program P so that any two destructive merges D_i and D_j are independent (Definition 47). The structure of program P is illustrated in Figure 4.6. It is easy to see that for any two destructive merges D_i and D_j

$$\texttt{reachable\_nodes}(D_i) \cap \texttt{reachable\_nodes}(D_j) = \{end\}.$$

Thus, using the sufficiency condition for independence (Lemma 5), any two destructive merges D_i and D_j are independent.

The profit p_i obtained by picking the item i will be mapped to the number of nodes which are *influenced* by the destructive merge D_i in P. Also, the weight of the item i will be mapped to the size of the *Region of Influence* of destructive merge D_i. Thus, the weight of item i corresponds increase in the code size which occurs when the corresponding split automaton a_i is applied.

The constraint of the total weight W of the knapsack is mapped to the increase in code size which we are allowed in SPLIT . Using the Additive Property (Theorem 6) for independent destructive merges, the total increase in code size is the sum of increases in code size due to the individual destructive merges. In particular, the increase in code size due to destructive merge D_i is $|\,\texttt{RoI}(D_i)\,|$.

It can be shown that split automaton a_i is in the optimal solution of SPLIT if and only if item i is selected in the optimal solution of KNAPSACK
. □

It is interesting to note that this hardness result does not rely on the complexity of the underlying data-flow analysis used, since we are already given the set of influenced nodes and the Region of Influence for each destructive merge. Further, the program P does not even contain any loops, and is acyclic. Thus, restricting the problem SPLIT any furhter does not result in a computationally tractable problem.

Example. Consider an instance of KNAPSACK with three items with the weights and profits as follows:

$$w_0 = 5,\ \ p_0 = 2,$$

$$w_1 = 3,\ \ p_1 = 1,$$

$$w_2 = 5,\ \ p_2 = 3.$$

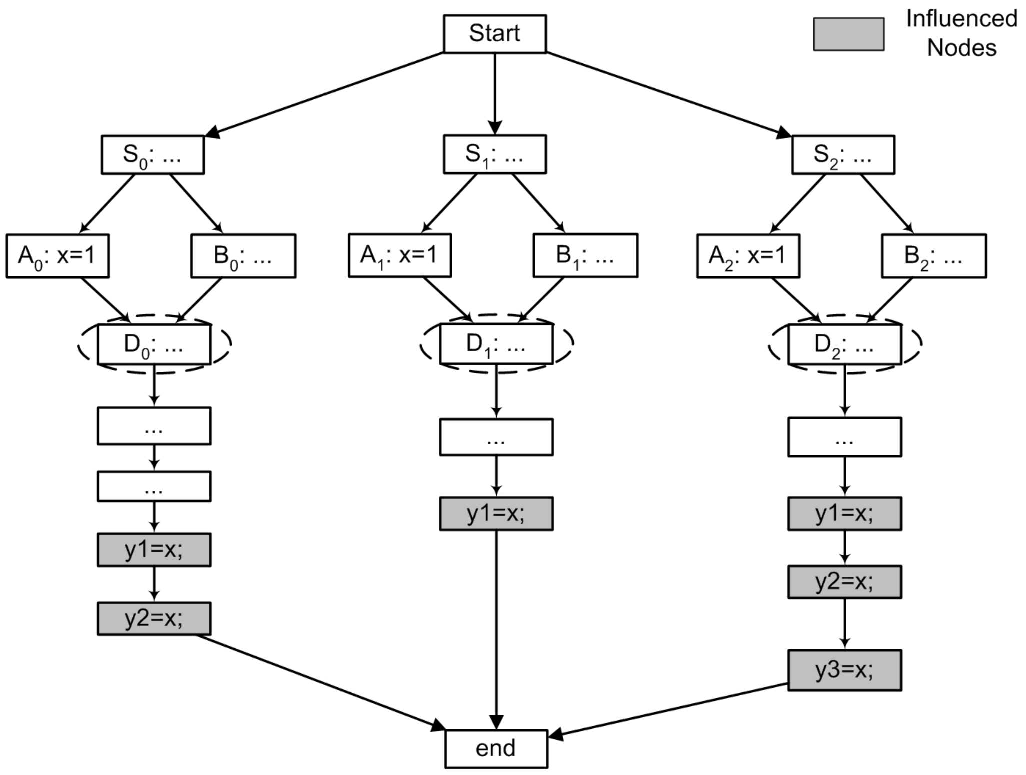


Figure 4.7: Program constructed for Constant Propagation corresponding to a particular instance of KNAPSACK . D_0, D_1 and D_2 are destructive merges.

```
FITNESS( $m$)
1  $profit \leftarrow count(m) * |\texttt{influenced\_nodes}(m)|$
2  $cost \leftarrow |\texttt{RoI}(m)|$
3  $fitness \leftarrow profit\ /\ cost$
4  **return** $fitness$
```

Figure 4.8: Computing the fitness of a destructive merge.

The corresponding program constructed is shown in in Figure 4.7. We see the corresponding to each item i in KNAPSACK , we have a destructive merge D_i with

$$|\ \texttt{RoI}(D_0)\ | = 5,\ \ |\ \texttt{influenced\_nodes}(D_0)\ | = 2,$$

$$|\ \texttt{RoI}(D_1)\ | = 3,\ \ |\ \texttt{influenced\_nodes}(D_1)\ | = 1,$$

$$|\ \texttt{RoI}(D_2)\ | = 5,\ \ |\ \texttt{influenced\_nodes}(D_2)\ | = 3.$$

Further, destructive merges D_0, D_1 and D_2 are all independent of each other.

□

4.2 Heuristic Solution

This lead us to device an aggressive greedy heuristic to solve this problem. Our approach is based on estimating the benefit obtained and cost incurred by eliminating a destructive merge. In the absence of profile information, we define the *fitness* of a destructive merge m to be

$$fitness(m) = |\texttt{influenced\_nodes}(m)|\ /\ |\texttt{RoI}(m)|.$$

Otherwise, we can make use of a low-cost basic-block profile to estimate the potential run-time benefit of eliminating a destructive merge. Let *count*(m) be the number of times the destructive merge was executed in the profile run. We now define the fitness to be

$$fitness(m) = count(m) * |\texttt{influenced\_nodes}(m)|\ /\ |\texttt{RoI}(m).$$

In this way, frequently executed destructive merges are more likely to be eliminated, and our approach can concentrate on the hot regions of code. Finally, we choose the k fittest destructive merges to be eliminated. It should be noted that while this heuristic method does not guarantee that the code size increase is within some bound (C), it works well in practice.

Figure 4.2 describes the algorithm for computing the fitness of a destructive merge m. Note that this algorithm applies to backward analysis as well.

Chapter 5

Related Work

Seek not to follow in the footsteps of wise men,
Seek what they sought.

Basho

5.1 Hot Path Graph Approach

An earlier proposal by Ammons and Larus [2] uses an acyclic path profile to try and improve the precision of the data-flow solution along hot paths. The approach consists of first using a Ball-Larus path profile [3] to determine the hot acyclic paths in the program. The next step in [2] consists of constructing a new CFG, called the *Hot Path Graph (HPG)*, in which each hot path is duplicated. This duplication is carried out in such a way that data-flow facts along the hot acyclic path do not get destroyed due to merge with facts along other overlapping acyclic paths. Conventional data-flow analysis is then carried out on the HPG. This approach relies on the assumption that removing control-flow merges along hot acyclic paths improves precision of data-flow analysis on hot path.

Consider the example code in Figure 5.1(a). Assume a path profile as shown in Table 5.1. Figure 5.1(b) shows the resulting HPG constructed assuming 100% coverage i.e. all taken paths are considered. Notice that in the HPG there are no control-flow merges along any of the acyclic paths listed in Table 5.1. For example, the two overlapping acyclic paths $B \rightarrow C \rightarrow E$ and $B \rightarrow D \rightarrow E$ in Figure 5.1(a) are separated into two separate paths $B1 \rightarrow C2 \rightarrow E2$ and $B1 \rightarrow D3 \rightarrow E3$ in the HPG. After performing conventional data-flow analysis on the HPG, the *use* of the variable a at node $B0$ can be replaced by the constant 0. However, the restructuring failed to optimize the two hot paths $B \rightarrow C \rightarrow E$ and $B \rightarrow D \rightarrow E$, and

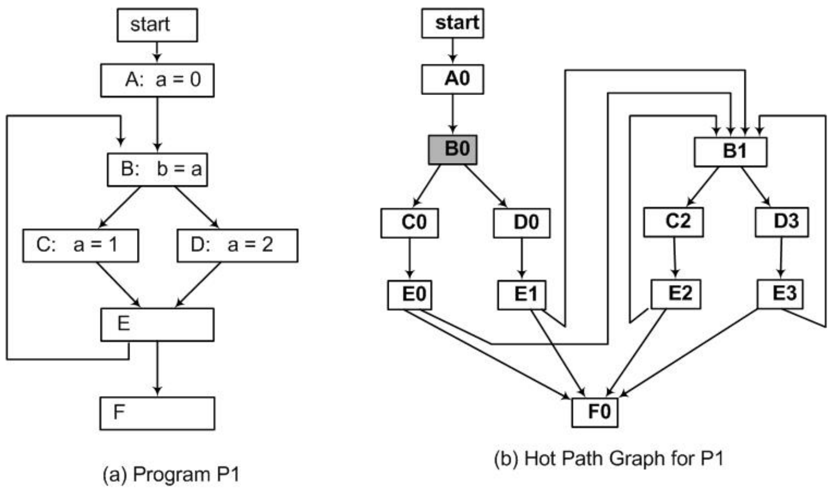


Figure 5.1: (a) Node E is a destructive merge. *Use* of variable a at node B cannot be replaced with a constant. (b)The *Hot Path Graph* corresponding to program $P1$. Node $B1$ is a destructive merge, and *use* of variable a still cannot be replaced by a constant.

Ball-Larus Acyclic Path	*Frequency*
$A \rightarrow B \rightarrow C \rightarrow E$	10
$B \rightarrow C \rightarrow E$	60
$B \rightarrow D \rightarrow E$	20
$B \rightarrow D \rightarrow E \rightarrow F$	10

Table 5.1: A path profile for the example in Figure 5.1. The frequency of the acyclic path denotes the number of times it was taken at run-time.

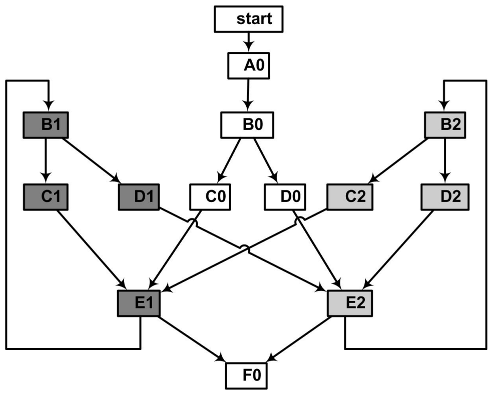


Figure 5.2: The *Split Graph* constructed from program $P1$ in which the *uses* of variable a at nodes $B0$, $B1$ and $B2$ can be replaced by constants 0, 1 and 2 respectively.

could not replace the *use* at node $B1$ with a constant value. The destructive merge E in the original CFG is removed in the HPG by duplicating code and creating two copies, $E2$ and $E3$. But the effect of the destructive merge has shifted to node $B1$, which is now a destructive merge since the data-flow facts $a = 1$ and $a = 2$ flowing along the incoming edges are merged at node $B1$. Thus, we see that simply duplicating acyclic paths does not always guarantee an increase in data-flow precision. Also, concentrating only on acyclic paths implies that all loop-back edges ($E2, B1$ and $E3, B1$ in the HPG) merge at a common loop-header (node $B1$ in the HPG). Thus, loop-headers which are destructive merges cannot be eliminated by the Ammons-Larus approach and data-flow precision is lost in these cases.

In comparison, the Split Graph constructed by our approach is shown in Figure 5.2. The destructive merge at node E is completely eliminated. In the Split Graph, *uses* of variable a at nodes $B0$, $B1$ and $B2$ can be replaced by constants 0, 1 and 2 respectively. Thus, we see that our approach effectively handles loop structures, guarantees additional optimization opportunities, and does not rely on expensive path profile information[1]. We compare the HPG method with our approach quantitatively in Chapter 6.

[1]For constructing the HPG the Ammons-Larus approach relies on the path profile information which is relatively more expensive than the simple basic block profile used in our Split Graph construction.

5.2 Other related approaches

In [7], an approach for complete removal of partial redundancy is described. Data-flow analysis is used to identify those regions of code which obstruct code motion. Code duplication and code motion are then used to eliminate the partial redundancy. Another approach targeted for PRE is discussed in [23]. Our approach is applicable for a more general class of data-flow problems as compared to these.

Code restructuring need not necessarily be limited to within a procedure. An extension of Ammons-Larus approach to the interprocedural case is described in [14]. A more recent framework [24] for whole-program optimization also considers code duplication to perform *area specialization*, which is purely profile-driven.

There have also been several other approaches which do not restructure the CFG in order to improve data-flow analysis precision. Holley and Rosen presented a general approach to improve data-flow precision by adding a finite set of predicates [12]. In [6], the precision of def-use analysis is improved by determining infeasible paths by using a low overhead technique based on detection of static branch correlations. Interestingly, path-sensitivity can also be obtained by synthesizing the name space of the data-flow analysis [4]. *Property simulation* is introduced in ESP [8] and is used to verify temporal safety properties. This approach keeps track of the correlation between "property state" and certain execution states. In [9], data-flow analysis is performed over a *predicated lattice*. The predicates used are determined automatically using a counterexample refinement technique. In [10], the context-sensitivity of the pointer analysis is adjusted based on the requirements of the client application. These approaches are complementary to the approach described in this paper.

Chapter 6

Experimental Results

I expected results.
BERNIE EBBERS,former CEO, WorldCom

In this chapter, we evaluate our approach for improving data-flow analysis precision. We have instantiated our framework for Constant Propagation and Liveness Analysis. We have implemented our approach in the Scale research compiler framework [21]. The framework is parameterised with the definition of a destructive merge, which depends on the data-flow analysis used, and on the definition of influenced nodes, which captures the interaction between the specific optimization and analysis.

6.1 Forward Analysis

We present experimental results for the specific problem of Constant Propagation [1]. We compare our approach (*Split*) with the Wegman-Zadeck conditional constant propagation algorithm [25](*Base*) and the Hot Path Graph approach(*HPG*) [2] using the SPECINT 2000 benchmark suite [22].

6.1.1 Benefits of Split Approach

We instantiate the constant propagation phase of the O1 pass of the Scale compiler with the default approach (*Base*), the *HPG* approach, and *Split*. The HPG approach uses a path profile generated using the *train* inputs for the respective programs, while the our Split approach uses a basic block profile from the same *train* inputs. The benchmarks are compiled for DEC ALPHA and were run on the 500MHz 21264 Alpha workstation. Running times were measures as average over multiple runs using the larger *ref* inputs.

Benchmark	*% speedup of Split over Base*	*% speedup of Split over HPG*
175.vpr	5	1
186.crafty	-2	2
197.parser	2	3
256.bzip2	0	3
300.twolf	3	-2
181.mcf	12	4
164.gzip	3	2
average	4	2

Table 6.1: Percentage speedup in the running times using Split in comparison to Base and HPG.

Benchmark	*Split / HPG*
175.vpr	1.15
186.crafty	1.10
197.parser	1.27
256.bzip2	1.11
300.twolf	0.93
181.mcf	13.75
164.gzip	2.32
average	3.5

Table 6.2: Ratio of the number of dynamic instructions with constant *uses* in Split over HPG.

Table 6.1 shows the speedup obtained by our Split approach over the Base approach and over the HPG approach. Split gives an average speedup of 4% over the Base case, and it gives an average speedup of 2% over the HPG approach.

To understand where the speedup comes from, we calculate the number of dynamic instructions which have constant *uses* indentfied by the restructuring transformation. This is computed by first performing constant propagation and replacing all constant *uses* in the original program. Restructuring (HPG or Split) is then carried out. The constant *uses* discovered can be attributed only to the restructuring. Thus, each instruction is weighted by the product of its execution count (using the *ref* inputs)and the number of new constant *uses*. The sum over all instructions gives us the number of dynamic instructions which have constant *uses* only because of restructuring. This metric has also been used in [2]. Table 6.2 shows the ratio of these instruc-

Benchmark	*Split / Base*	*Split / HPG*
175.vpr	1.5	0.7
186.crafty	2.0	0.7
197.parser	1.8	1.1
256.bzip2	1.9	0.5
300.twolf	2.0	0.7
181.mcf	1.9	1.0
164.gzip	1.5	0.8
average	1.8	0.65

Table 6.3: Ratio of code size increase of Split over Base, and of Split over HPG.

tions for Split over than of HPG. We observe an average of *3.5 times more* dynamic instructions with constants *uses* in Split as compared to HPG. In the 181.mcf Split results in as many as 13.75 times dynamic constant use instructions. This is because Split can handle cyclic structures effectively.

6.1.2 Cost of Split Approach

As mentioned earlier, the increase in precision comes at the cost of code duplication. We measured the code size in terms of the number of Scale intermediate instructions. Table 6.3 shows the ratio of the code size of Split over that of Base. We observe an average of *1.8× (80%) increase* due to Split. The Table also shows the ratio of code size of Split over that of HPG. We notice that Split incurrs less code size increase in comparison to HPG. Split shows an average of *0.65× (35%) decrease in code size* as compared to HPG.

6.2 Backward Analysis

We present experimental results for the specific problem of Liveness Analysis [1]. The experimental methodology used in the same as that for Constant Propagation. As before, we compare our approach, *Split*, with the baseline *Base*.

6.2.1 Benefits of Split Approach

We measure the benefits of our approach in terms of the percentage speedup obtained in comparison with Base.

Benchmark	*% speedup of Split over Base*
175.vpr	2
186.crafty	0
197.parser	1
256.bzip2	0
300.twolf	1
181.mcf	0
164.gzip	2
average	0.8

Table 6.4: Percentage speedup in the running times using Split in comparison to Base.

Benchmark	*Split / Base*
175.vpr	1.1
186.crafty	1.2
197.parser	1.1
256.bzip2	1.1
300.twolf	1.3
181.mcf	1.15
164.gzip	1.2
average	1.15

Table 6.5: Ratio of code size increase of Split over Base.

Table 6.4 shows the percentage speedup obtained. Improving the precision of liveness analysis causes code to become dead. We noticed that a majority of the code which became dead was not in loops. This is reflected in the lack up speedup we get for most benchmarks such as 186.crafty. On average, we get a average percentage speedup of 0.8%.

6.2.2 Cost of Split Approach

As before we measure the cost of the Split approach in terms of the increase in code duplication.

Table 6.5 shows the increase in code due to our code duplication. Due to the nature of the backward algorithm, the size of the Region of Influence is comparitively small. This is evidenced in the comparitively small increase in code size. On an average we see a 15% increase in code size.

Chapter 7

Conclusions and Future Directions

A conclusion is the place where you get tired of thinking.

ARTHUR MCBRIDE BLOCH

In this day, a man who says that something cannot be done, is apt to be interrupted by some idiot doing it.

ELBERT HUBBARD

We proposed a general framework to improve data-flow analysis precision based on restructuring the CFG of the program. The framework can be instantiated to any data-flow analysis. The actual transformation uses a known concepts of product automaton. We have proved that the transformation guarantees increase in optimization opportunities. Further, we showed that getting the optimal restructuring is NP-hard and proposed and evaluated a greedy heuristic. Our results indicate that our approach performs better than existing path profile driven approach [2].

7.1 Future Directions

7.1.1 Interprocedural Analysis

Our technique is currently restricted to be intra-procedural. It would be worthwhile to explore an inter-procedural extension to our approach along the lines of [14, 24], but which retains the simplicity and guarantees of our approach.

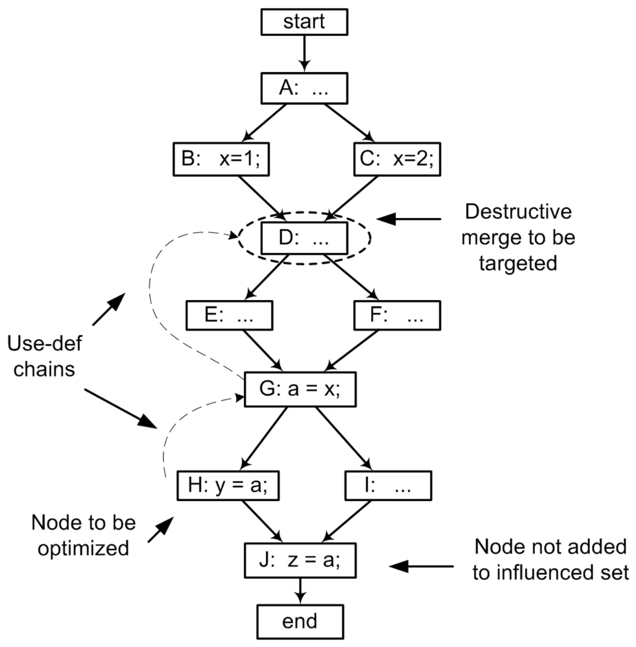


Figure 7.1: Figure illustrating a demand-driven version of our approach.

7.1.2 Points-to Analysis

Though as is our technique is applicable to points-to analysis, there are some interesting issues which arise when trying to make points-to analysis path sensitive. Specifically, the problem of estimating the *benefits* of improving precision at a program point for a given variable is not easy, since the benefit depends a lot on the client of the analysis.

7.1.3 Demand-driven Analysis

Currently, we target destructive merges where information is lost. We could also be given program statements at which we want better precision and carry out the restructuring in such a demand-driven fashion. For example, following the *use-def* chains of the variables at a statement we can trace back to the particular destructive merge which was responsible for the loss in precision. Further, the only influenced node for this destructive merge would be the one provided. The rest follows as described.

Example. Figure 7.1 illustrates this concept. Suppose we are told to optimize node $H : y = a;$. By following the *use-def* chains from $H \rightarrow G \rightarrow D$, we realise that the destructive merge D needs to be targeted. Setting the

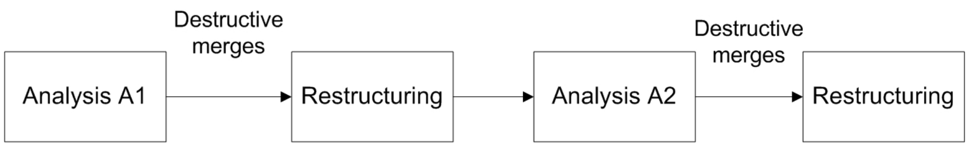


Figure 7.2: One possible way of handling restructuring for multiple analyses.

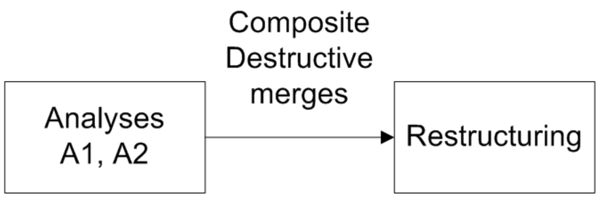


Figure 7.3: A composite approach to handle multiple analyses.

influenced nodes of D to the single node H we can carry out our restructuring to optimize H. Note that even though node J is also influenced by destructive merge D we do not include it in the influenced nodes.

□

7.1.4 Combined Restructuring for Multiple Analysis

Our approach can also be extended to handle multiple data-flow analysis in the same pass. For example, restructuring the control-flow graph only once to improve precision of constant propagation and availability analysis as shown in Figure 7.3, as opposed to a separate pass for each analysis as shown in Figure 7.2.

To achieve this we have to define the notion of a composite destructive merge taking into consideration both the analyses. This can be done in the following two ways.

Definition 49. (AND-COMPOSITE DESTRUCTIVE MERGE) *Given two analyses A_1 and A_2, a control-flow merge m is said to be a* and-composite destructive merge *if it is a destructive merge in A_1* and *in A_2.*

Alternatively, we could define a composite destructive merge as:

Definition 50. (OR-COMPOSITE DESTRUCTIVE MERGE) *Given two analyses A_1 and A_2, a control-flow merge m is said to be a* or-composite destructive merge *if it is a destructive merge in A_1* or *in A_2.*

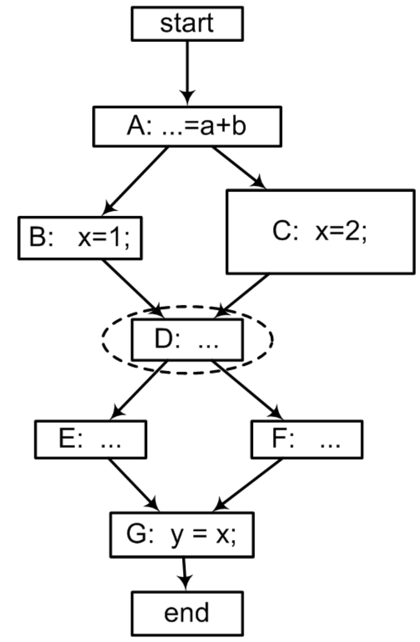


Figure 7.4: Program P2 used to illustrate composite destructive merge for constant propagation and availability analyses.

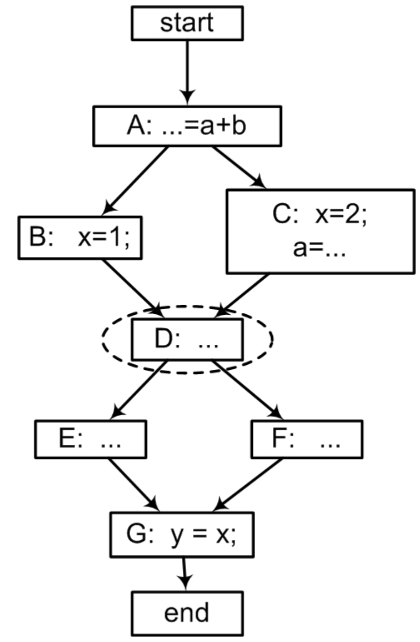


Figure 7.5: Program P1 used to illustrate composite destructive merge for constant propagation and availability analyses.

Example. Consider constant propagation and availability analyses. In Figure 7.4, merge D is *not* an and-composite destructive merge, since it is not a destructive merge for availability analysis. But in Figure 7.5, merge D is an and-composite destructive merge.

Note that in both the figures, merge D is an or-composite destructive merge.

□

In a similar vain, we need to define the notion of an influenced node in this composite analysis.

Definition 51. (AND-INFLUENCED NODE) *Given composite destructive merge m, a node n is said to an* and-influenced node *if it is an influenced node according to analysis A_1* and *analysis A_2.*

Definition 52. (OR-INFLUENCED NODE) *Given combined destructive merge m, a node n is said to an* or-influenced node *if it is an influenced node according to analysis A_1* or *analysis A_2.*

Having defined composite destructive merges and influenced nodes, we can perform the restructuring as described earlier by constructing a split automaton and performing a product. If we consider And-Composite Destructive merges and and-influenced nodes, then all the influenced nodes will be optimized in the restructured program. This can be seen as a generalisation of the Efficacy Theorem to this composite restructuring.

It is not clear as to whether such a combined restructuring will always give the same optimized program as the one obtained by performing separate passes. This is because the increase in precision of one analysis might result in different restructuring for the other other. More experimental and analytical evaluation needs to be carried out to better understand this.

7.2 Conclusions

In this thesis, we studied the problem of improving the precision of data-flow analysis. The approach we took was to restructure the control-flow graph of the program. Having studied the related literature, we found no uniform approach to tackle all kinds of data-flow analysis used commonly in compiler optimizations. We believe that the framework developed in this thesis conforms to the simplicity and generality constraints imposed by us. We have shown how to apply this to any forward and backward analysis. Specifically we illustrated our approach using constant propagation and liveness analysis.

Our claim of simplicity stems from the machinery we used to tackle this challenging problem. By using key ideas from automata theory, we could easily express our approach and also rigorously prove useful properties. Specifically, we used the product automaton to compute the restructured graph. We have methodically proved that our restructuring transformation is correct in that it preserves the semantics of the original program, and it is guaranteed to increase optimization opportunities in the restructured program.

Interestingly, having developed a framework for forward analysis, extending it to backward analysis posed interesting challenges. The approach described minimally changes the basic approach applicable to forward analysis and only appends the necessary changes to the approach.

The most difficult problem was to control the code explosion caused as a result of the code duplication. The major result here was to prove that finding the optimal restructured control-flow graph is NP-Hard. Furthermore, the proof also showed that the problem could not be simplified in order to make it tractable. This is the reason we developed a greedy algorithm. The algorithm makes use of profile information, though it does not completely rely on it.

Finally, we have implemented the framework into the Scale research compiler. Our results show promise, but as described in Section 7.1, there are a lot of interesting problems yet to be solved.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] Glenn Ammons and James R. Larus. Improving data-flow analysis with path profiles. In *PLDI*, pages 72–84, 1998.

[3] Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[4] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 237–251, 1998.

[5] Rastislav Bodik and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1997.

[6] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 361–377. Springer–Verlag, 1997.

[7] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, 1998.

[8] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[9] Jeffrey Fischer, Ranjit Jhala, and Rupak Mujumdar. Joining data flow with predicates. In *Foundations of Software Engineering*, pages 227–236, 2005.

[10] S. Guyer and C. Lin. Client-driven pointer analysis. In *International Static Analysis Symposium*, 2003.

[11] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[12] L. Howard Holley and Barry K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering (TSE)*, 7(1):60–78, 1981.

[13] Dexter C. Kozen. *Automata and Computability.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[14] David Melski and Thomas Reps. The interprocedural express-lane transformation. In *Compiler Construction*, 2003.

[15] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan-Kaufmann, San Fransisco, CA, 1997.

[16] Markus Müller-Olm and Oliver Rüthing. On the complexity of constant propagation. In *ESOP '01*, pages 190–205, London, UK, 2001. Springer-Verlag.

[17] Christos H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

[18] T. A. Proebsting. Proebsting's Law: Compiler Advances Double Computing Power Every 18 Years. https://research.microsoft.com/ toddpro/papers/law.htm. 1998.

[19] W. W. Pugh. Is Code Optimization (Research) Relevant?. http://www.cs.umd.edu/ pugh/IsCodeOptimizationRelevant.pdf.

[20] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *POPL '77*, pages 104–118, New York, NY, USA, 1977. ACM Press.

[21] Scale. A scalable compiler for analytical experiments. www-ali.cs.umass.edu/Scale/, 2006.

[22] SPEC. Standard Performance Evaluation Corporation. http://www.spec.org.

[23] Bernhard Steffen. Property-oriented expansion. In *Third Static Analysis Symposium*, pages 22–41, 1996.

[24] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *PLDI*, June 2006.

[25] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. In *ACM Transactions on Programming Languages and Systems*, pages 181–210, 1981.

Index